TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JUKKA HOLMI

NÄYTTÄMÖMEKANIIKAN OHJAUSSOVELLUKSEN KEHITTÄMINEN TUOTERUNKOARKKITEHTUURILLA

Diplomityö

Tarkastaja: Assistant Prof. David Hästbacka

Tarkastaja ja aihe hyväksytty
29. marraskuuta 2017

TIIVISTELMÄ

JUKKA HOLMI: Näyttämömekaniikan ohjaussovelluksen kehittäminen tuoterunkoarkkitehtuurilla
Tampereen teknillinen yliopisto
Diplomityö, 69 sivua
Marraskuu 2018
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Automaation tietotekniikka
Tarkastaja: Assistant Prof. David Hästbacka

Avainsanat: tuoterunkoarkkitehtuuri, tuoteperhe, piirremallinnus, SIMPLE

Tuoterunkoarkkitehtuuri on viimevuosikymmenten aikana helpottanut sekä tehostanut laajojen ja monimutkaisten ohjelmistojen kehitystyötä. Tuoteperheen yhteisen tuoterunkoarkkitehtuurin avulla sovellukset voidaan luoda ja julkaista lyhyemmässä ajassa, samalla säilyttäen korkeat laatuvaatimukset. Onnistunut tuoterunkoarkkitehtuuri edellyttää kuitenkin julkaistavilta tuotteilta keskinäistä samankaltaisuutta sekä mahdollisten tuotekohtaisten muutoksien tunnistamista ja hallintaa.

Tässä työssä tarkasteltiin tuoterunkoarkkitehtuurin mahdollisuuksia näyttämömekaniikan ohjaussovellusperheen kehittämisessä. Tavoitteena oli kehittää yrityksen nykyisen ohjaussovellusperheen arkkitehtuuria, analysoimalla tuotteiden välistä variaatiota sekä löytämällä tätä variaatioita arkkitehtuurissa edesauttavia suunnitteluratkaisuja. Lisäksi mahdollista tuoterunkoarkkitehtuuriin siirtymistä pohdittiin vertailemalla sitä nykyiseen tapaan toteuttaa tuoteperheen sovellukset.

Tuoteperheen muoto mallinnettiin tarkastelemalla kuutta jo luovutettua sovellusprojektia sekä määrittelemällä niiden pohjalta tuoteperhettä kuvaava piirremalli. Analyysin pohjalta saatiin käsitys siitä, mikä tuoteperheessä muuttuu, jonka jälkeen variaatiolle esitettiin suunnitteluratkaisuja. Ehdotettujen ratkaisujen toimivuutta testattiin demosovelluksen avulla. Testatut ratkaisut nojautuivat sovelluksen käynnistyksen yhteydessä sidottavaan, parametrien kautta määritettävään variaatioon.

Suunnitteluosiossa tarkasteltiin myös mahdollisia tuoterunkoon siirtymiseen soveltuvia keinoja. Proaktiivisen siirtymisen alkuinvestoinnille laskettiin alustava kustannusarvio, jossa vertailudatana toimi nykyisen menetelmän vaatima kehitysaika. Laskennassa käytettiin hyväksi tuoterunkoarkkitehtuuria varten kehitettyä laskentamallia. Mallin muuttujina käytetyt arviot perustuivat yrityksen menneiden projektien pohjalta luotuihin estimaatteihin.

Työssä havaittiin, että tietokantaan määritettyjen parametrien avulla dynaamisesti sidottavalla variaatiolla on mahdollista valjastaa ohjaussovellusperheestä tunnistettu variaatio tehokkaasti. Lisäksi tuoterungon siirtymisstrategiassa laskettu kustannusarvio antoi osviittaa mahdollisen tuoterunkoon siirtymisen perusteluille. Leikkauspisteeksi proaktiivisella menetelmällä osoittautui kuusi toimitettua sovellusta, jonka jälkeen tuoterunko olisi teoriassa nykyiseen menetelmään verrattuna kustannustehokkaampi.

ABSTRACT

JUKKA HOLMI: Enhancing stage control application with software product line architecture
Tampere University of Technology
Master of Science Thesis, 69 pages
November 2018
Master's Degree Programme in Automation Engineering
Major: Information Technology in Automation
Examiner: Assistant Prof. David Hästbacka

Keywords: software product line, software family, feature modeling, SIMPLE

For the last decades software product line engineering has been an emerging principle in the assembling of complex and large-scale software systems. Using a software product line shortens the time to market and at the same time maintains the needed high-quality standards. However, succeeding in software product line engineering requires a right amount of common and variable features to be found within the family of systems.

In this thesis the software product line engineering was studied against stage control application family, which are used to control stage machinery in a theater environment. The aim was to create more variable aware architecture by identifying the commonalities and variabilities found from the current applications. A possible adoption path to product line was also studied by modelling the economic impact of possible software product line and comparing it with the current method.

The current software family was analysed by comparing use cases and features of six already commissioned stage control applications. Based on the analysis, a feature model was gathered which showed the commonalities and variabilities within the stage control application family. Using the feature model, variable aware design techniques were proposed and tested. Implemented techniques realised the variation at the runtime start-up based on the set parameters.

The economic impact and possible approach of the product line was also studied. For the proactive approach the cost estimation was calculated by using a software product line cost estimation model and it was compared with the current method. The variables used in the cost model were based on real-life case estimations.

It was discovered that by using a parameter-based variation mechanism with dynamic binding it is possible to utilize the identified variation more easily. Also, the calculated cost estimation gave initial motivation for a possible product line approach. Results show that after six delivered applications the product line should be theoretically more efficient compared to current method.

ALKUSANAT

Haluan kiittää Insta Automation Oy:tä mahdollisuudesta tehdä tämä diplomityö. Aiheen keksimisestä sekä ohjaamisesta haluan kiittää kehitysosaston yksikönpäällikköä Arttu Hanhelaa. Kiitokset myös David Hästbackalle työn ohjaamisesta sekä loppuunsaattamisesta.

Haluan lopuksi kiittää avopuolisoani, perhettäni sekä ystäviäni tukemisesta sekä kannustuksesta koko opiskeluiden sekä tämän diplomityön ajalta.

Tampereella, 19.11.2018

Jukka Holmi

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

AOP	*Aspect Oriented Programming*, aspektipohjainen ohjelmistokehitys mahdollistaa läpileikkaantuvien ominaisuuksien modularisoimisen.
DIP	*Dependency Inversion Principle*, Riippuvuuksien kääntö -suunnitteluperiaate.
DRY	*Dont Repeat Yourself*, ohjelmistokehityksessä toistoa vähentävä suunnitteluperiaate, jonka mukaan ominaisuuksien tulisi esiintyä lähdekoodissa yksittäisinä abstraktioina.
IoC	*Inversion of control*, suunnittelumenetelmä ohjelman hallinnan siirtämisestä ulkopuoliselle kehykselle.
JSON	*Javascript Object Notation*, avoin tiedonmallinnustapa.
Node.JS	Javascript suoritusympäristö palvelinkoodin suorittamiseen.
NoSQL	*Not Only SQL*, relaationallisesta tietokannasta poikkeava tietokanta.
OPC UA	Teollisuusautomaation kommunikointiprotokolla.
REST	*Representational State Transfer*, web-palveluiden arkkitehtuurimalli.
ROI	*Return of investment*, pääoman tuottoaste.
SEI	*Software Engineering Institute*, on Yhdysvalloissa Carnegio Mellon yliopiston yhteydessä toimiva ohjelmistokehitykseen keskittynyt tutkimus ja kehitysyksikkö.
SIMPLE	*Structured Intuitive Model of Product Line Economics*, tuoterunkoarkkitehtuurin kustannuksien arviointiin kehitetty laskentamalli.
SOA	*Service Oriented Architecture*, palvelupohjainen arkkitehtuurimalli.
SOAP	*Simple Object Access Protocol*, web-palvelu viestintäprotokolla.
SOLID	5 olio-ohjelmoinnin suunnittelua ohjaavaa periaatetta.
UML	*Unified Modeling Language*, graafinen mallinnuskieli.

1. JOHDANTO

Tuotantolinjojen on tiedetty olevan valmistavassa teollisuudessa merkittävässä roolissa tehokkuuden parantajana jo 1900-luvun alun liukuhihnamenetelmistä lähtien. Hyvin rakennettu tuotantolinja mahdollistaa suorituskykyisen sekä kustannuksiltaan tehokkaan keinon luoda laadukkaita tuotteita asiakkaiden tarpeisiin. Tämä sama ajatus on viimevuosikymmenien aikana siirtynyt myös ohjelmistokehitykseen ja siinä tavoitteet ovat samat: tiettyä tuoteperhettä edustavat ohjelmistotuotteet halutaan julkaista markkinoille yhä lyhyemmässä ajassa, laadun kuitenkaan kärsimättä. Tällaisessa tapauksessa samaa tuoteperhettä edustavat ohjelmistotuotteet voidaan luoda tehokkaasti käyttämällä hyväksi tuoteperheelle yhteistä tuoterunkoarkkitehtuuria.

Whisper Show Control on näyttämömekaniikan toiminnanohjaussovellus, joka toimii ylemmän tason hallintajärjestelmänä näyttämömekaniikan koneautomaatiolle. Kyseessä on ohjelmistotuote, joka räätälöidään projektikohtaisesti asiakkaan laitteistoon sekä vaatimuksiin nähden. Sovelluksen avulla voidaan suorittaa lavasteita liikuttavien koneiden ohjaustoimenpiteitä sekä rakentaa monimutkaisempia, usean laitteen suorittamia ajosekvenssejä. Käyttäjä pystyy sovelluksen avulla itsenäisesti määrittelemään laitteilla suoritettavat liikkeet sekä rakentamaan näiden pohjalta esityksen mukaisen lavasteiden ohjauksen.

Jokainen toimitettu toiminnanohjaussovellus sisältää piirteitä, jotka ovat sovelluksesta riippumatta samanlaisia sekä piirteitä, jotka muuttuvat asiakkaan vaatimia ominaisuuksia sekä laitekantaa mukaileviksi. Toiminnanohjaussovellus toimitetaan asiakkaalle yleensä osana teatteriprojektikokonaisuutta ja sen ominaisuudet räätälöidään vastaamaan toimitettua laitekantaa. Teatterijärjestelmä on modulaarinen kokonaisuus niin mekaniikan kuin myös ohjausjärjestelmän suhteen.

Piirteiden aiheuttama muutos edellyttää käytetyltä sovellusrungon arkkitehtuurilta oikeanlaista joustavuutta, jotta muutostyö onnistuisi projektin kehitysvaiheessa tehokkaalla tavalla. Nykyisin käytössä olevan sovellusrungon arkkitehtuuri ei kuitenkaan mahdollista muutoksen edellyttämää modulaarisuutta. Ohjelmiston arkkitehtuuri tulisikin miettiä uudelleen niin, että sovelluksen kehitys ja ylläpito olisi tulevaisuudessa ajallisesti sekä kustannuksiltaan tehokkaampaa. Eräs keino tähän on määritellä ohjelmistoperheelle tuoterunkoarkkitehtuuri, joka määrittelee yhteisen ydinarkkitehtuurin sekä keinot sovelluskohtaiseen erikoistamiseen.

1.1 Arkkitehtuurin kehittäminen

Bass *et al.* mukaan ohjelmiston arkkitehtuuri kattaa yleisesti ohjelmiston korkealla tasolla ilmenevän kokonaisvaltaisen rakenteen, sekä rakenteen sisältä löytyvien osakokonaisuuksien rakenteet, näiden rakenteiden väliset vuorovaikutukset ja ominaisuudet [1]. Ohjelmiston arkkitehtuurin voidaan myös sanoa olevan joukko strategisia suunnittelupäätöksiä, jotka vaikuttavat lopputulokseen. [2, s.37] Painottamalla tiettyjä laatuvaatimuksia päädytään tietynlaisiin suunnittelupäätöksiin, jolloin onnistuneen arkkitehtuurin voidaankin sanoa olevan optimaalinen ratkaisu painotettuihin laatuominaisuuksiin nähden. Arkkitehtuurin suunnittelua ohjaavat sovellukselle esitetyt vaatimukset. [1] Niin toiminnalliset, kuin myös ei toiminnalliset vaatimukset asettavat tavoitteita sille, millainen arkkitehtuurin tulisi olla. Lisäksi vaatimukset riippuvat myös sen esittäjästä, sillä eri sidosryhmillä on erilaiset toiveet esimerkiksi siitä, mitä arkkitehtuurin kautta tulisi saavuttaa tai kuinka paljon arkkitehtuurin toteuttaminen saisi mahdollisesti kustantaa.

Diplomityön kohteena olevan sovellusrungon elinkaari ulottuu kymmenien vuosien taakse. Se on pitkän historiansa aikana saavuttanut optimaalisen tilanteen tiettyjen laatuominaisuuksien osalta: esimerkiksi suorituskyky on hioutunut käyttäjän kannalta optimaaliseksi ja asiakkaan vastaanottama valmis tuote on lopullisessa muodossaan yleensä hyvin luotettava sekä toiminnaltaan vakaa. Pitkä historia on kuitenkin tuonut mukanaan sen, että sovellusrunko on kerryttänyt elinkaarensa aikana ominaisuuksien toteutustavoista johtuen teknistä velkaa. Teknistä velkaa syntyy yleensä esimerkiksi siinä vaiheessa, kun sovelluksen ominaisuuksia luodaan nopealla aikataululla, jolloin toteutuksessa painotetaan ylläpidettävien ratkaisujen sijaan nopeutta [3].

Minimoimalla kehityksessä käytettyä aikaa, ominaisuuksien toteutuksessa hyödynnetään helposti opportunistisia lähestymistapoja, kuten esimerkiksi kopio & liitä tekniikkaa [4]. Näin ollen teknistä velkaa kerryttäen, sovelluksen lähdekoodi saavuttaa tietyssä vaiheessa pisteen, jossa ominaisuudet läpileikkaavat toisiaan eikä selkeää rakennetta ei ole enää helppo tunnistaa. Tällöin tekninen velka on saavuttanut niin korkean pisteen, että jo pienien muutoksien tekemisessä joudutaan käyttämään hyvin paljon aikaa. Pahimmillaan tekninen velka aiheuttaa teknisen konkurssin, jolloin koko sovellus joudutaan suunnittelemaan sekä toteuttamaan alusta asti uudelleen uusien ominaisuuksien aikaansaamiseksi [3]. Teknisen velan ottaminen ei aina välttämättä ole tietoinen päätös ja toisinaan sen ottaminen voi olla jopa eduksi: teknistä velkaa ottamalla joissain tapauksissa saada nopeutettua markkinoille pääsyä. [5] Velan ottamisesta täytyisi kuitenkin olla tietoinen ja samalla sitä ottaessa tulisikin miettiä miten velka saadaan maksettua tulevaisuudessa takaisin.

1.2 Tavoitteet

Nykyisellään lähdekoodirungon valjastaminen toimintakuntoon projektista toiseen vaatii jokaisessa tapauksessa suuren manuaalisen lähdekoodin muokkaustyön, joka on luonteeltaan aikaa vievää sekä virheherkkää. Esimerkiksi tietyt yksittäiset ominaisuudet läpileikkaavat lähdekoodissa useammassa paikassa, jolloin näihin ominaisuuksiin kohdistuvat muutokset aiheuttavat kertaantuvan työmäärään. Sovellus sisältää myös paljon vahvoja riippuvuussuhteita käytettyihin sovelluskomponentteihin, jolloin komponenttien vaihtaminen ei onnistu, ilman merkittävää työmäärä. Lisäksi lähdekoodirungon valjastaminen toimintakuntoon edellyttää korkeaa tietämystä liittyen koko ohjausjärjestelmän sekä sovelluksen toiminnallisuuteen, joka asettaa korkean kynnyksen sovelluksen kehittäjälle. Edellä mainittujen ongelmakohtien korjaamiseksi, sovelluksen arkkitehtuuria sekä lähdekoodia tulisi näin ollen modernisoida modulaarisemmaksi.

Modulaarisempi arkkitehtuuri todennäköisesti edesauttaisi positiivisella tavalla sovelluksien kehitystyötä, sillä eri kokonaisuudet sekä vastuualueet olisivat selkeämmin tunnistettavissa. Lopulta arkkitehtuurin kehittyminen vaikuttaisi positiivisesti myös sovelluksen loppukäyttäjiin, sillä asiakkaiden mahdolliset lisätoiveet saadaan modulaarisessa arkkitehtuurissa toteutettua helpommin. Haasteena on tunnistaa, millä tavoin uusi arkkitehtuuri tulisi suunnitella ja tehdä, jotta nykyisen sovellusrungon tarjoamat laatuvaatimukset säilyisivät. Lisäksi arkkitehtuurin modernisoinnin tulisi olla taloudellisesti perusteltua.

Koska nykyisessä sovellusrungossa kysymyksessä selkeästi tunnistettavan toimialan sovellus, jossa muuttuvat sekä staattiset piirteet ovat toimitettujen järjestelmien välillä tunnistettavissa, on varteenotettava strategia modulaarisen arkkitehtuurin kehittämiseen tuoterunkoarkkitehtuuri. Tuoterunkoarkkitehtuurissa ajatuksena on systemaattinen resurssien uudelleenkäyttö saman tuoteperheen sisäisten sovelluksien kesken ja sen tarkoituksena on edistää pitkällä tähtäimellä sovelluskehityksen tehokkuutta sekä laatua [6-8].

Onnistumisen edellytyksenä on modulaarinen ja joustava arkkitehtuuri, joka mahdollistaa tuoteperheen sisällä tunnistettujen muunneltavuusaspektien realisoinnin yksiselitteisellä ja ketterällä tavalla. Tuoterunkoarkkitehtuuri ottaa konseptina huomioon teknologian lisäksi myös sovelluksen kehitysprosessiin sekä liiketoimintaan liittyviä näkökulmia.

1.3 Tutkimusongelma ja menetelmät

Tässä diplomityössä tarkoituksena on tutkia tuoterunkoarkkitehtuurin soveltuvuutta kohdesovellusperheen modernisoinnin vaihtoehtona sekä kartoittaa erilaisia vaihtoehtoja tuoteperheen muunneltavuuden huomioon ottavan arkkitehtuurin toteutukseen. Tavoitteena on saada nykyistä sovellusarkkitehtuuria kehitettyä niin, että tuotteen projektikohtainen muunneltavuus onnistuisi tulevaisuudessa tehokkaammin.

Tutkimusprosessi etenee niin, että ensiksi kirjallisuuskatsauksen avulla selvitetään tuoterunkoarkkitehtuurin liittyvä uusin tekniikka. Tämän jälkeen tutkimusongelmaa lähdetään ratkaisemaan mallintamalla tuoteperheen jo julkaistujen sovelluksien välinen keskinäinen samankaltaisuus ja muunneltavuus. Tunnistettuihin piirteisiin nojautuen etsitään tämän soveltuvia käytännön ratkaisumenetelmiä. Työn tutkimuskysymykset on määritelty seuraavanlaisiksi:

- Millä tavoin olemassa olevan tuoteperheen muunneltavuus ilmenee ja miten se voidaan mallintaa?

- Millä tavoin tunnistetut muunneltavuuspisteet tulisi mahdollisesti käytännössä toteuttaa tuoterunkoa ajatellen?

- Millä tavoin mahdollinen siirtyminen tuoterunkoarkkitehtuuriin tulisi toteuttaa ja millä tavoin siirtyminen olisi taloudellisesti perusteltua?

Kirjallisuuskatsaus alkaa luvulla 2, jossa käydään läpi tuoterunkoarkkitehtuuriin liittyvää käsitteistöä, joihin kuuluvat mm. tuoterunkoarkkitehtuurin kehitysprosessimalli, tuoterunkoarkkitehtuurin eri siirtymisstrategiat sekä eri tavat mallintaa tuoterunkoarkkitehtuurin muunneltavuus. Lisäksi tarkasteluun otetaan tuoterunkoarkkitehtuurin taloudellinen näkökulma sekä menetelmään liittyviä riskejä. Luvussa 3 esitetään yksityiskohtaisemmin erilaisia ohjelmistoteknisiä ratkaisuja tuoterunkoarkkitehtuurin muunneltavuuden toteuttamiseen.

Suunnitteluosassa (luku 4) olemassa olevalle tuoteperheen sovelluksille tehdään ominaisuuksien kartoitus, jonka pohjalta luodaan tuoteperheen variaatiota kuvaava piirremalli. Piirremalliin pohjautuen esitetään sittemmin suunnitteluratkaisuja, jotka ottavat tunnistetun variaation huomioon. Ehdotettuja menetelmiä testataan toteuttamalla variaation huomioon ottava, rajatun toiminnallisuuden sisältävä demo-sovellus. Lopuksi tarkastellaan potentiaalisia mahdollisuuksia tuoterunkoon siirtymisessä.

Luvussa 5 työn tulokset koostetaan yhteen ja niitä arvioidaan vasten muissa julkaisuissa sekä kirjallisuudessa esiintyviä tuloksia. Lisäksi kappaleessa esitetään alustavat johtopäätökset sekä mahdolliset jatkokehitysajatukset.

2. TUOTERUNKOARKKITEHTUURI

Tässä luvussa kerrotaan tuoterunkoarkkitehtuurin konseptista sekä siitä, millä tavoin menetelmää voidaan hyödyntää osana tuoteperheen sovelluskehitystyöprosessia. Tarkasteluun otetaan menetelmän taloudellinen kuin myös variaation, eli muunneltavuuden huomioiva näkökulma.

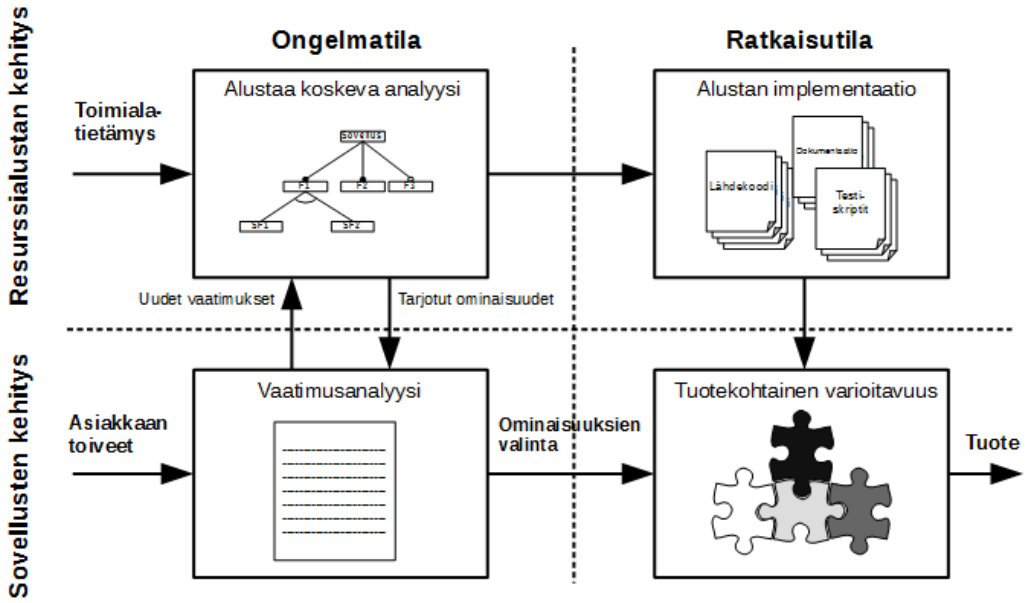
Tuoterunkoarkkitehtuuri on viime vuosikymmenten aikana voimistunut konsepti tuoteperheen sisäisestä suunnitelmallisesta uudelleenkäytöstä [6-8]. Menetelmän ydinajatuksena on jakaa sekä uudelleen käyttää tuoteperheen sisällä yhteistä muunneltavissa oleva resurssikantaa. Yksittäisen tuotteen erikoistaminen tehdään yhdistelemällä sekä parametrisoimalla saatavilla olevia tuoteresursseja soveltuvin tavoin. Tavoitteena on tukea tuoteperheen sisällä tehokkaampaa sekä laadukkaampaa ohjelmistokehitystä. Tuoterunkoarkkitehtuuri käsittää näin ollen joukon sovelluksia, jotka jakavat yhteisen, hallitun joukon yhteisiä piirteitä, jotka palvelevat tietyn markkinasegmentin tarpeita ja jotka on kehitetty yhteisistä ydinkomponenteista, suunnitellulla tavalla [8]. Kyseessä voidaan siis ajatella olevan konsepti suunnitelmallisesta, skaalautuvasta ohjelmistoarkkitehtuurista, joka mahdollistaa tehokkaalla tavalla tuoteperheeseen kuuluvan sovelluksen kehittämisen.

Uudelleenkäytettävät resurssit eivät synny kuitenkaan ilman huolellista suunnittelutyötä. [8] Siksi resurssialustan realisoiminen vaatiikin tarkkaa etukäteissuunnittelua sekä näkemystä tuoteperheen tulevasta elinkaaresta ja suunnasta. Suunnittelussa tulee ottaa huomioon tuoteperheen sisällä esiintyvä variaatio sekä samankaltaisuus. Lisäksi resurssien suunnittelussa tulisi huomioida mahdolliset tulevaisuuden kehityssuunnat. Onnistunut tuoterunkoarkkitehtuuri vaatiikin sitoutumista niin kehittäjien, kuin myös organisaation johdon puolelta.

2.1 Kehitystyöprosessi

Tuoterunkoarkkitehtuurissa kehitystyöprosessin ajatellaan jakautuvan kahteen erilliseen osaan, resurssialustan kehitystyöprosessiin sekä sovelluskehitystyöprosessiin [7, 8].

Niin resurssialustan, kuten myös sovelluskehitystyöprosessin tarkoituksena on kuvan 1 tavoin tunnistaa vaatimusten kautta syntyvä ongelmatila ja löytää siihen soveltuvin ratkaisutila. [9, luku 2.2] Resurssialustan suunnittelussa tärkeään rooliin asettuu toimialatietämys, kun taas sovelluskehityksessä tärkeintä on vastaanottaa asiakkaiden toiveet sekä tunnistaa ne tuoterungon tarjoamat resurssit, jotka voisivat näitä toiveita palvella. Mahdolliset asiakkaiden esittämät uudet vaatimukset, joita alusta ei vielä nykyisessä muodossaan tarjoa, välitetään resurssialustan kehitysosastolle, jolloin heidän tulee ratkaista, miten alustan muunneltavuus saadaan noudattamaan myös näitäkin toiveita.

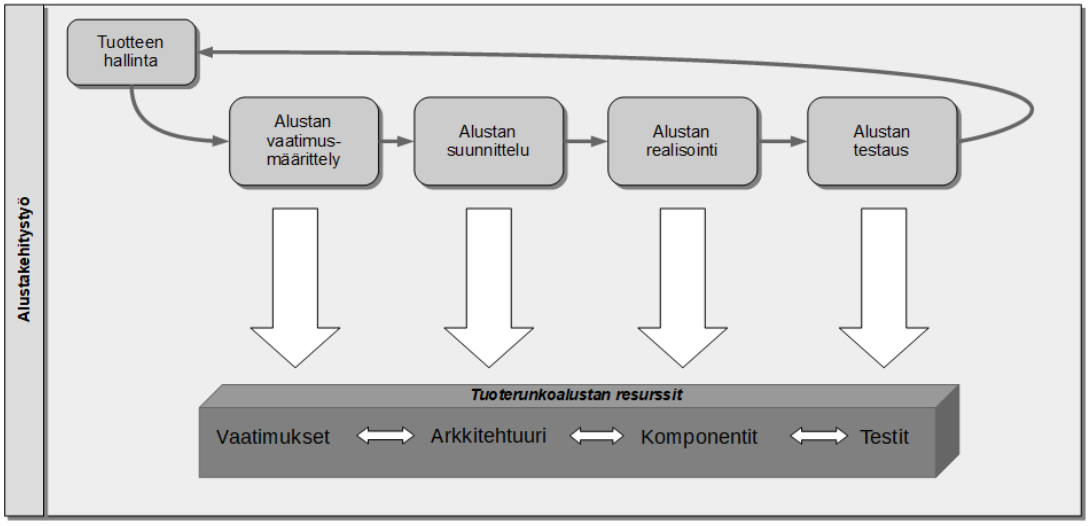


Kuva 1. *Tuoterunkoarkkitehtuurin kehitysprosessin jakautuminen. Mukailtu lähteestä [9, luku 2.2].*

Eriyttämisen kautta saavutettu etu mahdollistaa sen, että molemmilla puolilla voidaan keskittyä omiin vahvuusalueisiin: resurssialustan kehittäjien tavoitteena on luoda mahdollisimman geneerinen alusta, joka mahdollistaa tehokkaalla tavalla tuotekohtaisen muunneltavuuden. Sovelluskehittäjien tehtävänä on taas sijaita lähellä asiakasrajapintaa, luoda resurssialustan pohjalta toimiva sovellus sekä tuoda tarvittavat uudet vaatimukset resurssialustan kehitystiimin tietoon [7].

2.1.1 Resurssialustan kehitystyöprosessi

Kuva 2 esittää yksityiskohtaisemmin ne aliprosessit, joista resurssialustankehitysprosessi koostuu. [7, s.21] Kokonaisuus muodostuu tuotteenhallinnasta, alustan vaatimusmäärittelyprosessista, alustan suunnitteluprosessista, alustan realisointiprosessista sekä alustan testausprosessista.



Kuva 2. *Tuoterunkoarkkitehtuurin kehitystyöprosessin kaavio. Mukailtu lähteestä [7, s.24].*

Tuotteen hallinta on vastuussa tuotteen kaupallisesta hallinnoinnista. [7, s.24-25] Tuotehallinnassa tehdään päätökset siitä, mitä osia tuoterunkoarkkitehtuurissa lähdetään toteuttamaan. Päätöksenteon herätteenä tuotehallinnalle toimii ylemmän johdon asettamat tähtäimet, eteenpäin lähtevänä herätteenä annetaan tuoteperheen etenemissuunnitelma, jonka pohjalta määritellään tuoterungon piirteet sekä tulevat mahdolliset ominaisuudet. Tuotteen hallinnan kautta on tarkoitus muodostaa näkemys tuoterungon tuotevalikoimasta [8].

Alustavaatimusmäärittely aliprosessin tavoitteena on määritellä tuotteen etenemissuunnitelman pohjalta muunneltavuutta kuvaava malli tuoterungosta. [7, s.25-26] Tuoteperheen sisällä esiintyvät samankaltaiset ja muuttuvat piirteet tunnistetaan ja mallinnetaan. Tätä kautta saadaan kuva resurssialustan vaatimuksista. Vaatimusten pohjalta hahmottuvat myös tuotteita koskevat rajoitteet [8].

Alustan suunnittelu -aliprosessissa tehtävänä on suunnitella tuoterungon referenssiarkkitehtuuri. [7, s.26] Referenssiarkkitehtuuri tarjoaa korkean tason rakennemallin rungon pohjalta luotaville sovelluksille. Tavoitteena on saada aikaiseksi alusta, joka tukee mahdollisimman hyvin tuoterungolle asetettuja muunneltavuusvaatimuksia. Referenssiarkkitehtuuri voidaan esittää esimerkiksi useamman näkökulman huomioon ottavan Kruchen 4+1 menetelmän mukaisesti [10].

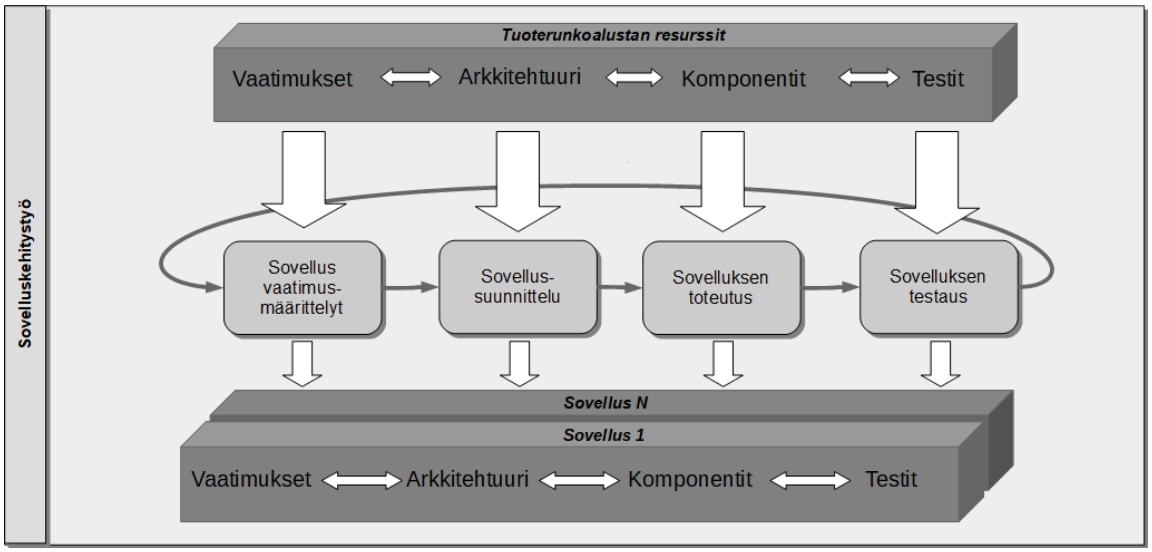
Alustan realisointi -aliprosessi käsittää yksityiskohtaista suunnittelua seuraavan käytännön toteutuksen. [7, s.27] Tässä aliprosessissa syntyvät ne uudelleenkäytettävät sovelluskomponentit ja resurssit, joita sovelluskehittäjät voivat hyödyntää tuoteperheen sovelluksien rakentamisessa. Jokaisen tässä aliprosessissa syntyvän resurssin tulisi olla suunniteltu uudelleenkäyttöä ajatellen.

Alustan testaus -prosessissa resurssialustan tarjoamat ohjelmistokomponentit validoidaan ja testataan vasten niille asetettuja vaatimuksia. [7, s.27] Alustan testausprosessin tavoitteena luoda joukko uudelleenkäytettäviä testausresursseja. Nämä voivat olla mm. uudelleenkäytettäviä testikomponentteja sekä testausstrategioita.

Resurssialustan kehitystyöprosessin tuloksena tulisi syntyä kokonaiskuva resurssialustan tarjoamista resursseista, tuoterungon tuotevalikoimasta sekä mahdollisesta tuotteiden tuotantosuunnitelmasta [8].

2.1.2 Sovelluskehitystyöprosessi

Sovelluskehitysprosessissa tavoitteena on erikoistaa tuoterunkoalustan pohjalta itse ohjelmistotuote käyttämällä hyväksi alustan tarjoamaa muunneltavuutta [7, 8].



Kuva 3. *Sovelluskehitystyöprosessin aliprosessit. Mukailtu lähteestä [7, s.31].*

Sovelluskehitystyöprosessi koostuu kuvan 3 mukaisesti neljästä eri aliprosessista: vaatimusmäärittelyprosessista, suunnitteluprosessista, toteutusprosessista sekä testausprosessista [7, s.22].

Vaatimusmäärittelyprosessissa tavoitteena on tunnistaa ja määritellä alustan pohjalta luotavan sovelluksen vaatimukset. [7, s.31-32] Tavoitteena on löytää mahdollisimman kattavasti ne alustan tarjoamat resurssit, joita voitaisiin hyödyntää sovelluksen rakentamisessa.

Sovelluksen suunnitteluprosessissa suunnitellaan asiakkaan käyttöön tuleva sovellus tuoterunkoalustan referenssiarkkitehtuuria hyväksi käyttäen. [7, s.32-33] Suunnitteluprosessissa resurssialusta alustetaan ja konfiguroidaan vastaamaan yksittäisen sovelluksen vaatimuksia. Sovelluksen suunnitteluprosessissa on tärkeää noudattaa tuoterungon sovelluksien yhteistä suunnitelmaa, joka voi olla muodoltaan referenssiarkkitehtuurikuvaus tai jokin muun kehitystä tukeva ohjeistus.

Sovelluksen toteutusprosessissa luodaan konkreettinen sovellus pohjautuen referenssisuunnitelmaan [7, s.33]. Maksimaalinen hyöty saadaan silloin, mikäli sovellus pystytään rakentamaan täysin resurssialustan pohjalta, ilman räätälöintiä. [9, luku 2.2.4] Tehokkuusnäkökulmaa edesauttaa myös se, mikäli resurssialustan tarjoaa mahdollisuuden sovelluksen erikoistamisen automaattisesti. Tällaisessa ideaalitapauksessa ainoa sovellukseen kohdistuva manuaalinen työ muodostuu siitä, että sovelluksen rakentaja valitsisi mukaan integroitavat ominaisuudet, jonka jälkeen sovelluksen toiminnallisuus generoidaan valintojen mukaisesti.

Sovelluksen testausprosessissa lopullinen sovellus validoidaan ja verifioidaan vasten sille asetettuja vaatimuksia. Resurssialustan tulisi tarjota valmiita työkaluja myös yksittäisten sovelluksien testauksen suorittamiseen. [7, s.33-34]

2.2 Taloudellinen näkökulma

Tuoterunkoarkkitehtuurissa taloudellista etua saadaan käyttämällä hyväksi tuoteperheen yhteistä resurssikantaa [11]. Onnistunut tuoterunkoarkkitehtuuri parantaakin parhaimmillaan yhtä aikaa sekä kehitystyön tuottavuutta, että sovelluksien laatua. [8, 12] Onnistuneen tuoterungon ehtona on kuitenkin se, että organisaatio on valmis tekemään tarvittavia muutoksia sekä investointeja tuoterungon hyväksi.

Ennen käytännön soveltamista, tuoterunkoarkkitehtuurista voidaan luoda kuva toteuttamalla sen ympärille liiketoimintatapaus (*Business Case*). Liiketoimintatapauksen mallintamisen avulla on mahdollista saada arvio siitä, kuinka realistisena tuoterunkoarkkitehtuuriin siirtymistä voidaan pitää [12, s.191]. Mallinnetun liiketoimintatapaus toimii lopullisessa muodossaan päätöksentekoa tukevana dokumentaationa, jonka avulla tuoterunkoon siirtymistä voidaan perustella. Tavoitteet voivat liittyä esimerkiksi laadun kehittämiseen, uusien markettisegmenttien valtaamiseen, sovelluspäivitysten yksinkertaistamiseen, nopeampaan kehitysaikaan tai asiakaskunnan laajentamiseen [8].

Bosch on määritellyt liiketoimintatapauksen muodostamisen vaiheet seuraavalla tavalla [12, s.192]:

Analysoidaan nykyinen tilanne, eli tunnistetaan jo käytössä olevan lähestymistavan heikkoudet ja vahvuudet. Tavoitteena on saada tilannekuva nykyisen lähestymistavan vaatimista resursseista, sekä siitä, miten paljon esimerkiksi ylläpitoon, uusien ominai-

suuksien lisäyksiin ja tuotteen kokonaisvaltaiseen kehitykseen joudutaan nykyisellään käyttämään vaivaa.

Luodaan pitkän tähtäimen ennustus nykyisellä menetelmällä, eli luodaan arvio kustannuksista suhteessa tavoiteltuihin ominaisuuksiin. Nykyistä tilannetta analysoimalla saadaan yleensä riittävä kuva siitä, saavutetaanko sitä hyödyntämällä tulevaisuudessa tavoiteltavat tähtäimet. Taipuuko nykyinen menetelmä esimerkiksi tulevaisuudessa tuoteperheeseen kohdistuviin vaatimuksiin? Pitkän tähtäimen ennustuksessa saadaan samalla myös arvio tarvittavasta resurssoinnin tarpeesta, jonka muutos mahdollisesti edellyttää.

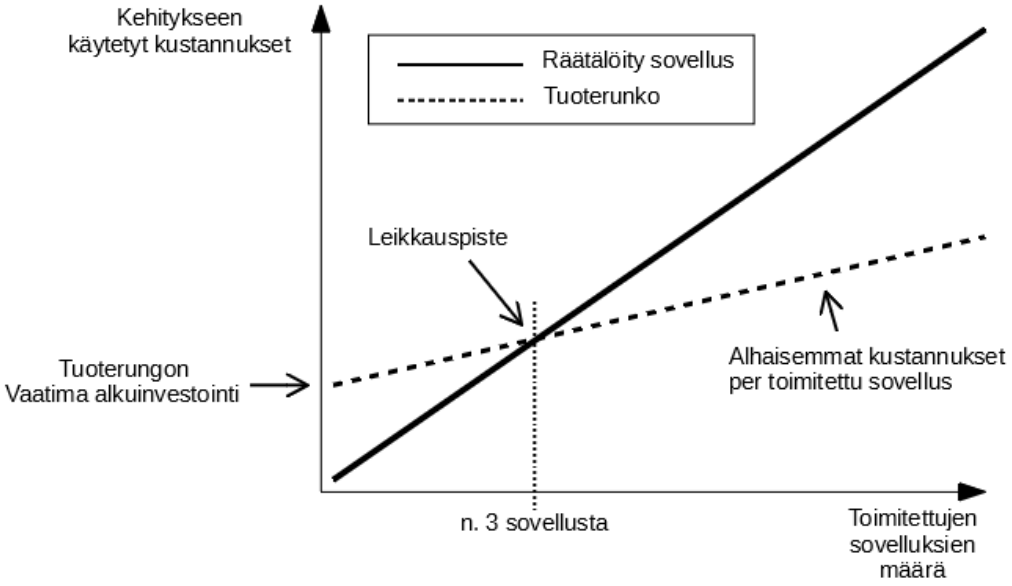
Tuoterunkoon siirtymistä koskevan investointianalyysin tekeminen. Luodaan ennustus syntyvistä kuluista ja eduista, mikäli lähestymistavassa siirryttäisiin tuoterunkoarkkitehtuuriin. Kuinka paljon muutos tulisi kokonaisuudessaan maksamaan?

Luodaan pitkän tähtäimen ennustus koskien tuoterunkoarkkitehtuurin soveltamista. Kun tuoterunkoon siirtymiseen tarvittavat kustannukset on tunnistettu, on mahdollista luoda pitkän tähtäimen ennustus tuoterunkoarkkitehtuurin soveltamisesta. Ennustuksen kautta on mahdollista saada kuva siitä, saavutetaanko tuoterunkoarkkitehtuurilla halutut tavoitteet ja mitä tavoitteiden saavuttaminen kaiken kaikkiaan kustantaa.

Yhteenveto. Edellä mainitut kohdat läpikäymällä on mahdollista saavuttaa yhteenveto kunkin lähestymistavan hyödyistä ja haitoista. Yksinkertaisimmillaan se lähestymistapa, joka tarjoaa suurimman katteen voittaa, mutta kyseessä ei välttämättä ole näin suoraviivainen asia: tuoterunkoarkkitehtuurin käyttöönottoon liittyy yleensä alkuvaiheessa enemmän riskejä, liittyen esimeriksi uuden teknologian käyttöönottoon. Riskit kuitenkin pienentyvät tuoterungon kypsyessä.

Hyvin määritelty liiketoimintatapaus toimii itsessään tuoterunkoalustan resurssina. [8] Sen vuoksi sitä olisi hyvä kehittää samalla kun tuoterunkoarkkitehtuuri kehittyy. Tällä tavoin voidaan ylläpitää jatkuvaa analyysiä tuoterungon liiketoimintaa koskevasta tulevaisuudesta.

Liiketoimintatapauksen investointianalyysissa, kustannuksien arviointiin voidaan hyödyntää tuoterunkoarkkitehtuuria varten kehitettyjä laskentamalleja. Laskentamallien avulla on esimerkiksi mahdollista löytää kuvan 4 tavoin teoreettinen leikkauspiste, jonka jälkeen tuoterungon käyttäminen on kustannustehokkaampaa [11].



Kuva 4. *Leikkauspiste, jonka jälkeen tuoterunkoarkkitehtuuri on räätälöityyn sovelluskehitykseen verrattuna kannattavampi menetelmä. Mukailtu lähteestä [11, s.4].*

Tapauskohtainen leikkauspiste saadaan laskettua käyttämällä laskentamalleissa tapaukseen soveltuvia sovelluskehityksen kuluarvioita tai kustannuksien estimointeja [13]. Laskentamallien avulla voidaan arvioida tuoterunkoarkkitehtuurin kautta syntyviä kuluja ja hyötyjä sekä tehdä osana liiketoimintatapausta analyysia siitä, voisiko tuoterunkoarkkitehtuurin siirtyminen olla taloudellisesti kannattavaa [14].

Laskentamalleja on useita, joista maininnan arvoisia ovat esimerkiksi Poulin esittämä tuoterunkoarkkitehtuuria käsittävä ROI laskentamalli [15], Boehm *et al.* julkaisussa [16] esittämä COPLIMO, eli *Constructive Product Line Investment Model*, sekä viimeisempänä, SEI:n kehittämä SIMPLE eli *Structured Intuitive Model of Product Line Economics* [14].

SIMPLE esittää kustannusfunktioiden sekä hyötyfunktioiden kautta taloudellisen mallin tuoterunkoarkkitehtuurin muodostamista kustannuksista. SIMPLE-mallin toiminta koostuu käytännössä neljästä eri kustannusfunktiosta:

- $C_{org}(t)$: palauttaa kustannukset siitä, kuinka paljon organisaation siirtyminen tuoterunkoarkkitehtuuriin kokonaisuudessaan maksaa. Kyseiset kustannukset voivat koostua, esimerkiksi uudelleenorganisoinnista, sovelluskehitysprosessien muutoksesta tai tuoterunkoalustan vaatimasta henkilöstön koulutuksesta.

- $C_{cab}(t)$: palauttaa arvion siitä, kuinka paljon tuoterunkoalustan yhteisten resurssien luominen kokonaisuudessaan kustantaa. Funktio ottaa huomioon kaiken sen työmäärän, joka resurssien luomiseen liittyy, kuten tuoteperheiden piirteiden mallintamisen sekä tuoterunkoalustan komponenttien ohjelmoinnin.

- $C_{unique}(t)$: palauttaa kustannusarvion siitä, kuinka paljon kuluja syntyy tuotekohtaisten uniikkien ominaisuuksien lisäämisestä.

- $C_{reuse}(t)$: palauttaa kustannusarvion siitä, kuinka paljon sovelluksen rakentaminen maksaa käyttäen hyväksi tuoterungon yhteisiä resursseja.

Kustannusfunktioiden yhdistelmästä syntyy kaava (1), jonka avulla voidaan arvioida tuoterunkoarkkitehtuurin kokonaiskustannuksia:

$$C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n}(C_{unique}\,(product_i, t) + C_{reuse}(product_i, t)) \quad (1)$$

SIMPLE ei ota kantaa, millä tavoin kustannusfunktioiden tai hyötyfunktioiden muoto on käytännössä toteutettu. [8] Sen sijaan malli kertoo yleisen muodon ja jättää sen soveltajalle vapauden määritellä funktioiden sisäinen toteutus itse.

Erilaisten taloudellisten mittareiden laskentaan SIMPLE tarjoaa useita valmiita laskentamalleja. Esimerkiksi kaavassa (2) on esitetty laskentamalli, jonka avulla voidaan arvioida tuoterunkoarkkitehtuurin kautta saatava säästö verrattuna räätälöityyn sovelluskehitykseen.

- $C_{prod}(t)$: palauttaa räätälöidyn sovelluskehityksen vaatiman kustannuksen.

$$\sum_{i=1}^{n} C_{prod}(product_i, t) -$$

$$(C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n}(C_{unique}\,(product_i, t) + C_{reuse}(product_i, t))) \quad (2)$$

ROI:n, eli pääoman tuottoasteen laskentaan SIMPLE tarjoaa (3) mukaisen yhtälön.

$$ROI = \frac{\sum_{i=1}^{n} C_{prod}(product_i) - [C_{org}() + C_{cab}() + \sum_{i=1}^{n}(C_{unique}(product_i) + C_{reuse}(product_i,))]}{C_{org}() + C_{cab}()} \quad (3)$$

SIMPLE-malli kertoo puhtaasti lukujen pohjalta tuoterunkoarkkitehtuurin kannattavuudesta, eikä se ota kuitenkaan huomioon muita tuoterunkoarkkitehtuurin tuomia positiivisia vaikutuksia, kuten esimerkiksi luotettavuuden sekä laadun kehittymistä [11]. Nämä ominaisuudet realisoituvat yleensä tuoterunkoarkkitehtuurin resurssien kypsyessä tarpeeksi.

2.3 Tuoterunkoon siirtyminen

Mikäli yritys haluaa siirtyä soveltamaan tuoterunkoarkkitehtuuria, tulee siirtymiselle määritellä soveltuvin strategia. Siirtymisstrategian määrittely kuuluu osaksi resurssialustan tuotantostrategiaa [14]. Siirtymisstrategian valintaan vaikuttaa yrityksen tilanne tuoteperheen suhteen sekä investointihalukkuus. Charles Krugerin mukaan siirtymis-

strategia luokitellaan kolmeen eri kategoriaan: proaktiiviseen, ekstrakstiiviseen sekä reaktiiviseen [9, 17].

Proaktiivisessa lähestymistavassa, joka tunnetaan myös termillä *Big Bang* -strategia [7, s.397] tuoterunkoalustan resurssit koostetaan kerralla kasaan ennen yhdenkään tuotteen julkaisua. Tämä lähestymistapa vaatii yleensä korkeimman alkuinvestoinnin ja se sisältää korkeimman riskin, mutta sen pohjalta tuoterunkoalustan arkkitehtuurista saadaan juuri sellainen kuin halutaan. Proaktiivinen siirtyminen nähdään yleensä akateemisessa mielessä ideaalisena keinona tuoterunkoarkkitehtuuriin siirtymisessä, mutta johtuen sen vaatimasta korkeasta alkuinvestoinnin tarpeesta sekä oikeanlaisen arkkitehtuurin määrittämisestä heti alussa, ei se välttämättä ole pienempien tai keskisuurten yrityksen kannalta paras mahdollinen keino [9, 18, 19]. Lisäksi, mikäli proaktiivisen menetelmän soveltamisen aikana todetaankin, että tuoterunkoarkkitehtuurin siirtyminen olikin kokonaisuudessa epäonnistunut idea, syntyy kustannuksia menetetystä ajasta sekä rahasta huomattava määrä [7, s.400].

Ekstraktiivisessa lähestymistavassa jo olemassa olevia sovelluskomponentteja käytetään hyväksi tuoterunkoarkkitehtuurin resurssialustan kokoamisessa. Kokoamista ei tarvitse tehdä yhdellä kertaa vaan resursseja voidaan siirtää tuoterunkoalustan käyttöön inkrementaalisesti. Tämä lähestymistapa soveltuu parhaiten yrityksille, joilta löytyy sovelluksia joltain aikaisemmalta tai olemassa olevalta toimialalta. Ekstraktiivisessa lähestymistavassa ongelmana on vanhan koodin aiheuttama taakka, mikäli sitä ei ole toteutettu tuoterungon muunneltavuutta edellyttävällä tavalla. Tämä saattaakin aiheuttaa ongelmia ylläpidon kannalta, mikäli tuoterunkoalusta kasvaa tarpeeksi. [9, 18]

Reaktiivinen lähestymistapa on myöskin luonteeltaan inkrementaalinen ja se soveltuu parhaiten tilanteisiin, joissa uuden tuoterunkoalustaa hyödyntävän sovelluksen vaatimukset ovat vielä ennustamattomissa. Kyseessä on siis ketterä menetelmää noudattava siirtymästrategia. Liikkeelle lähdetään tilanteesta, jossa resurssialustan vaatimia resursseja luodaan ensiksi yhden julkaistavan tuotteen tarpeisiin. Tästä tuotteesta resursseja louhitaan sittemmin alustan sekä muiden tuotteiden käyttöön. Kysynnän kasvaessa uudelleen käytettäviä resursseja syntyy kehityksen myötä lisää ja näin ollen kovin suurta alkuinvestointia, jonka esimerkiksi proaktiivinen lähestymistapa vaati, ei tarvitse tehdä heti tuoterungon aloitusvaiheessa. [9, 18]

On muistettava, että reaktiivisessa sekä myös ekstraktiivisessa -menetelmässä louhitut resurssit eivät kata pelkästään ohjelmakoodiin liittyviä resursseja, vaan resurssi voi luonteeltaan olla mitä tahansa, joka edesauttaa tuoteperheen sovelluksen syntyä. Resurssit voivatkin olla puhtaan koodin sijaan esimerkiksi liiketoimintamalleja, säännöstöjä, algoritmeja, prosessimäärittelyjä tai suorituskykymalleja [8].

2.4 Tuoterunkojen muunneltavuus

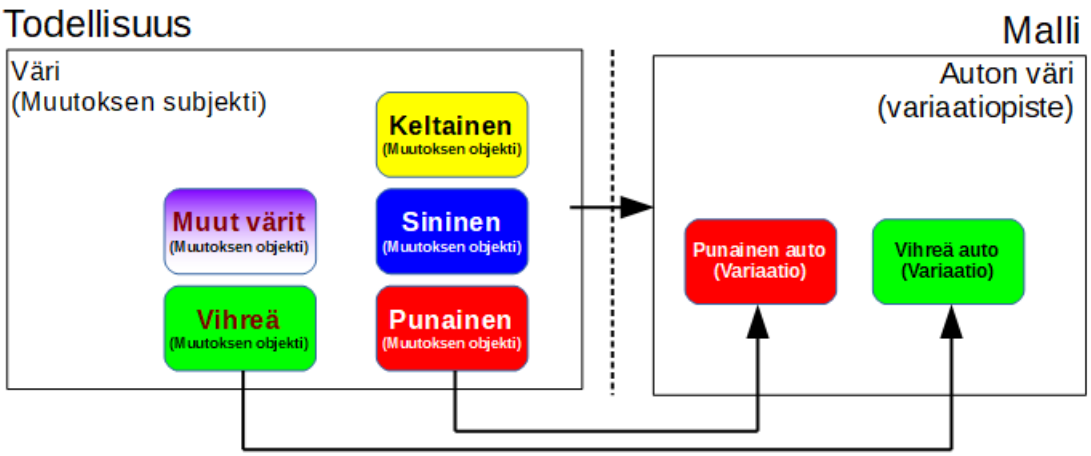
Tuoterunkoarkkitehtuurin resurssialustan tehokkaan hyödyntämisen edellytyksenä on, että sovelluskohtainen muunneltavuus eli variaatio on tunnistettu oikealla tavalla. Tämän vuoksi tuoterunkoarkkitehtuuria suunnitellessa tuoteperheen muunneltavuus tulee mallintaa jollakin tavoin. Hyvin dokumentoitu muunneltavuus edesauttaa muun muassa päätöksentekoa, kommunikointia sekä jäljitettävyyttä [7].

2.4.1 Muunneltavuuden tunnistaminen

Miten muuttuvat piirteet tulisi sitten tunnistaa tuoterunkoarkkitehtuurin kontekstissa? Pohl *et al.* käyttävät reaalimaailmassa esiintyvän variaation kuvaamiseen seuraavanlaisia termejä: muutoksen subjekti ja muutoksen objekti. [7, s.60] Muutoksen subjekti viittaa todellisuudessa vallitsevan piirteeseen, kun taas muutoksen objekti käsittää piirteen instanssin, eli ilmentymän. Esimerkki muutoksen subjektista voisi olla väri ja kyseisen muutoksen objekteja ovat kaikki mahdolliset näkyvän aallonpituuden värit.

Muutoksen subjektin pohjalta voidaan luoda käsite paremmin tuoterunkoarkkitehtuurin kontekstiin sopivasta termistä nimeltä variaatiopiste. [7, s.61] Variaatiopiste käsittää muutoksen subjektit tuoterunkoarkkitehtuurin rajoissa. Eli mitkä asiat ovat tuoterungossa mahdollisesti muunneltavissa. Koskimiehen mukaan, variaatiopiste käsittää myös muunneltavuusvaatimuksen mahdollistavan suunnitteluratkaisun [20, s.172]. Variaatiopisteestä ilmentyvää vaihtoehtoa voidaan taasen kutsua nimellä variaatio, joka käsittää muutoksen objektit tuoterunkoarkkitehtuurin rajoissa.

Esimerkki edellä mainitusta on esitetty kuvassa 5. Auton väri olisi teoriassa mahdollista värjätä minkä väriseksi tai kuvioiseksi tahansa. Kuitenkin esimerkin tuoterunko tarjoaa auton väriä koskevana variaationa ainoastaan vaihtoehdot, punainen ja vihreä. Tämä voi johtua esimerkiksi siitä, että näillä väreillä on asiakkaiden keskuudessa suurin kysyntä, jolloin tuoterungon on liiketoiminnallisista syistä järkevintä tarjota ensisijaisesti nämä värivaihtoehdot.



Kuva 5. *Variaatiopisteen sekä variaation suhde muutoksen subjektiin ja objektiin. Mukailtu lähteestä [7, s.63].*

Tuoterungon variaatio voidaan luokitella sillä tavoin, että onko kysymyksessä ulkoisesti vai sisäisesti ilmentyvästä variaatiosta. [7, s.68] Sisäisesti muuttuvalla variaatiolla tarkoitetaan sovelluksen kehittäjille ilmentyvää muutosta, jonka suora näkyvyys on piilossa sovelluksen käyttäjältä, kun taas ulkoinen variaatio on suoraan sovelluksen käyttäjän tunnistettavissa sekä vaikutuksen alaisena. Esimerkki ulkoisesta variaatiosta voisi olla käyttäjän mahdollisuus valita, millä kielellä hän haluaa sovelluksen käyttöliittymän toimivan, kun taas sisäinen variaatio määrittelee, mitä muunnostoimenpiteitä ohjelman sisäisessä toiminnassa joudutaan tekemään kielimuutoksen realisoimiseksi.

Sisäinen variaatio kumpuaa pitkälti ulkoisen variaation kautta, sillä monet sovellukselta vaaditut ominaisuudet aiheuttavat sovelluksen syvimmissä kerroksissa muutosta ja sitä kautta sisäistä variaatiota. [7, s,69] Mitä syvemmälle muutoksen toteuttamisessa mennään, sitä enemmän työtä muutoksen realisointi tuottaa ja sitä vähemmän työmäärä suoraan näkyy suoraan loppukäyttäjälle. Sisäisen ja ulkoisen variaation mukainen rajanveto toimii kuitenkin asiakkaan näkökulmasta paremmin: asiakkaan on helpompi tehdä päätöksiä nojautuen rajattuun määrään erilaisia vaihtoehtoja [7, s.70]. Varsinkin, jos variaatiot ovat hyvin teknisorientoituneita, ei asiakas välttämättä osaisi ottaa kantaa siihen, mihin ominaisuuksiin hänen kannattaisi ensisijaisesti vaikuttaa.

Niin sisäinen, kuten myös ulkoinen variaatio synnyttävät tuoterungon kontekstissa avaruudellisen muutokset (*variability in space*), eli eri tuotevariaatioiden kautta ilmentyvän muutoksen [7, s.66]. Kysymyksessä on siis tuoterungon pohjalta julkaistujen, erilaisten konfiguraation sisältävien tuotteiden synnyttämästä muutoksesta. Tämä muutos on tunnistettava osana resurssialustan arkkitehtuurin suunnittelua, jotta eri tuotteiden välillä esiintyvät ominaisuuksien vaihdannaisuudet saataisiin tehokkaalla tavalla toteutettua.

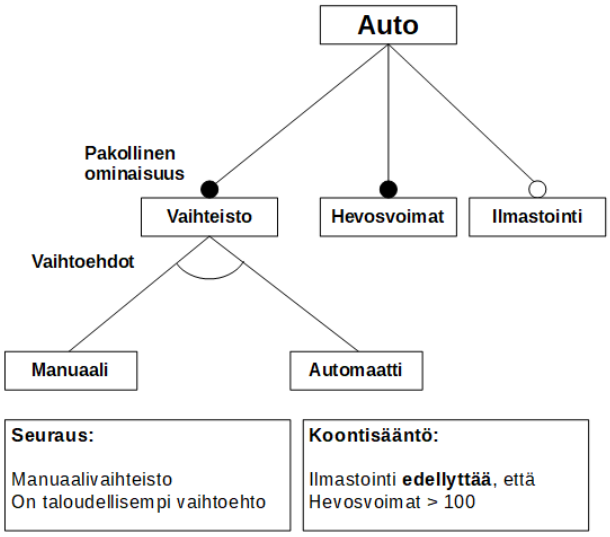
Väistämätöntä on myös se, että ohjelmistoihin kohdistuu niiden elinkaaren aikana versiomuutoksia. Kysymyksessä on tällöin ajan suhteen (*variability in time*) tapahtuvasta

muutoksesta. [7, s.65] Tuoterungon variaatiopisteessä voidaan huomioida ajan suhteen tapahtuva muutos ottamalla huomioon esimerkiksi teknologinen kehitys. Näin ollen ominaisuuksista ja niiden rajapinnoista voidaan rakentaa sellaisia, että nykyaikaan päivittäminen onnistuu tarvittaessa helposti. Ajan suhteen tapahtuva muutoksen voidaan myös ajatella tapahtuvan eri sovelluskerroksissa erilaisina aikoina. Esimerkiksi käyttöliittymäkerros kokee todennäköisemmin tiheämmin muutoksia, verrattuna vaikka sovelluksen tietokantakerrokseen [21, s.67].

Niin muutoksessa ajan kuin myös muutoksessa tilan suhteen avaintekijäksi muodostuu oikeanlainen sovellusarkkitehtuuri sekä konfiguraation hallinta, joka mahdollistaa ajan ja tilan suhteen ilmenevien muutoksien tapahtuvan hallitulla tavalla.

2.4.2 Muunneltavuuden mallintaminen

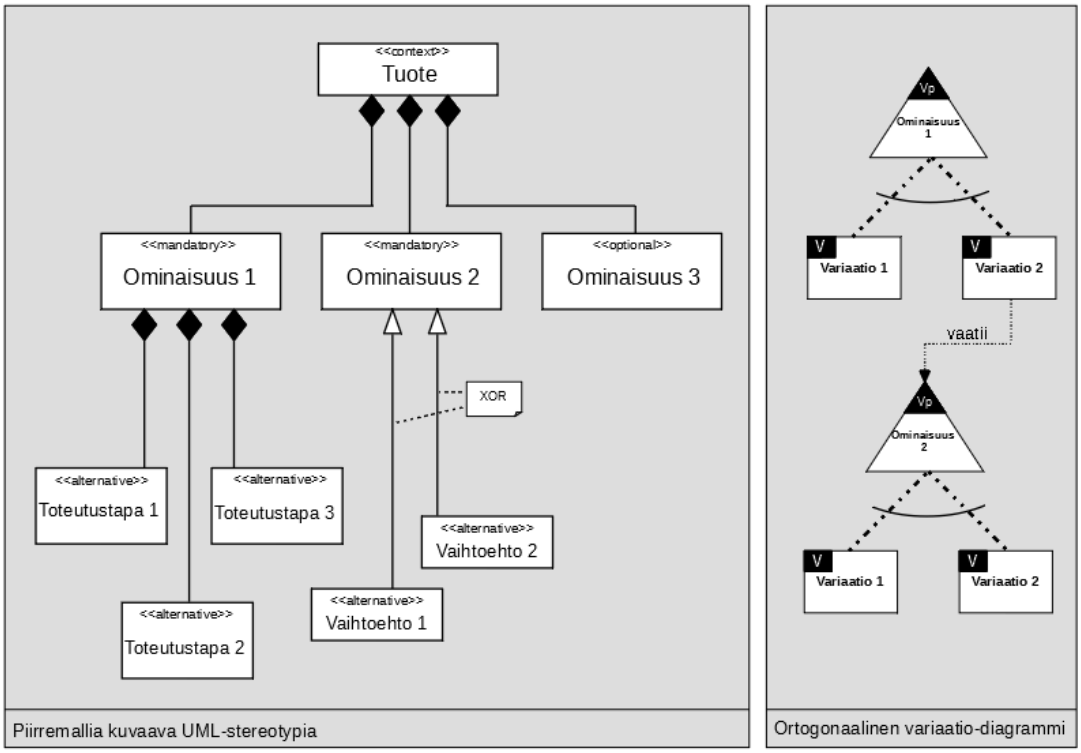
Tuoterungon kontekstissa muunneltavuus voidaan koostaa yhteen kuvaamaan koko tuoteperheen käsittävää variaatiota sekä eri variaatioiden välisiä riippuvuuksia. Eräs käytetty tapa mallintaa muunneltavuus on piirremalli, jossa kuvataan tuoteperheen yhteiset sekä vaihtoehtoiset ominaisuudet [22]. Kuvassa 6 on esitetty esimerkki piirremallin käyttämisestä auton ominaisuuksien havainnollistamisessa.



Kuva 6. *Autoa kuvaava piirremalli. Mukailtu lähteestä [22].*

Piirremallissa nivoutuvat yhteen eri variaatiot sekä näiden väliset riippuvuussuhteet. Myös rajoitteet sekä muut mallin kuvausta tukevaa informaatiota voidaan esittää osana piirremallin toteutusta. Esimerkiksi kuvassa 6 nähdään, että auton ilmastointi edellyttää hevosvoimia löytyvän enemmän kuin 100. Tällainen informaatio tukee piirremallin omaa kuvausta lisäämällä siihen tarvittavia lisäehtoja tai seurauksia.

Muitakin tapoja piirteiden mallintamiseen on, kuten ominaisuusmatriisi [12, s.197], sekä kuvassa 7 esitettävät UML-stereotypian piirremalli [20, 23] ja ortogonaalinen piirremalli [7]. Ortogonaalisen piirremallin esitystapa on tarkoitettu kuvaamaan puhtaasti tuoterunkoarkkitehtuurin muunneltavuutta ja sen avulla voidaan esittää samalla myös peilaus suunnittelutason ratkaisuun [24]. Kuvan 6 mukainen piirremalli on kuitenkin tällä hetkellä yksi käytetyimpiä tapoja mallintaa tuoteperheen variaatio [9, luku 2.3].



Kuva 7. *Variaation mallintaminen UML-stereotypialla sekä ortogonaalisella piirremallilla. Mukailtu lähteistä [7, 20].*

2.4.3 Muunneltavuuden sitomishetki

Tunnistettu muunneltavuus realisoidaan osaksi sovellusta jossain vaiheessa sen elinkaarta. Muunneltavuuden luonne määrittelee pitkälti sen, millä tavoin ja missä vaiheessa sitominen voidaan käytännössä tehdä. Sitomishetken voidaan esimerkiksi määritellä tapahtuvan joko arkkitehtuuritasolla, yksityiskohtaisen suunnittelun tasolla tai toteutustasolla [20, s.210].

Toteutustason jälkeen esiintyvät sitomisajankohdat voidaan yleistäen jakaa kolmeen eri kategoriaan, sitomiseen käännösvaiheessa, sitomiseen sovelluksen käynnistyessä sekä sitomiseen ohjelman suorituksen aikana [9, 25].

Käännösvaiheessa tapahtuva muunneltavuuden sitominen edesauttaa suorituskyvyn sekä resurssien optimointia, sillä ylimääräistä koodia ei sisällytetä osaksi suoritettavaa so-

vellusta. Tämä kuitenkin aiheuttaa samalla sen, että sovellus ei kääntämisen jälkeen ole enää muunneltavissa vaan mahdolliset muutokset vaativat koko koodin uudelleenkäännön [9, luku 3.1.1].

Käynnistyksen yhteydessä tapahtuva muunneltavuuden realisoiminen tapahtuu luonnollisesti ohjelman kääntämisen jälkeen ohjelman käynnistyksen yhteydessä. [9, luku 3.1.1] Tällaisessa lähestymistavassa ohjelma mukautetaan toimimaan variaation edellyttämällä tavalla muuttamalla sovelluksen moduuleja erikoistamisparametrien mukaisesti.

Ohjelman suorituksen aikana tapahtuva sitominen mahdollistaa varioitavuuden sovelluksen käydessä. [9, luku 3.1.1] Tällöin ohjelmaa ei tarvitse kääntää tai käynnistää uudestaan muutoksen astuessa voimaan. Tämän kaltainen muunneltavuus esiintyy esimerkiksi käyttäjän muokattavissa olevissa käyttöliittymässä sekä dynaamisissa tuoterungoissa, joissa järjestelmä kykenee mukautumaan dynaamisesti vallitseviin olosuhteisiin [26].

Esimerkiksi jo edellisessä aliluvussa esitetyssä kuvitteellisessa sovelluksessa eräs muunneltavuuspiste oli kielen vaihto sovelluksen käytön aikaisesti. Näin ollen variaation tulisi olla mahdollista toteuttaa dynaamisesti ilman sovelluksen uudelleenkääntöä. Täysin vastakohtainen esimerkki voisi olla jokin turvallisuuskriittinen sovellus, jossa koodin muunneltavuus on toteutettava ennen käännösvaihetta, sillä turvallisuuskriittisessä sovelluksessa muistihallinnan tulee olla luonteeltaan staattista [27]. Sovelluksen luonne määritteleekin pitkälti sen, miten ja missä vaiheessa variaation sitomoinen käytännössä tehdään.

2.5 Tuoterunkojen riskit

Tuoterunkoarkkitehtuuri on yksittäiseen sovelluskehitykseen verrattuna monimutkaisempi kokonaisuus, jonka vuoksi siihen sisältyy myös enemmän riskejä [11]. Laajuutensa takia riskit myös koskevat useampia eri sidosryhmiä.

Tuoterunkoarkkitehtuurissa kehitystyöprosessi jaettiin kahteen osaa, jotta ongelmia voitaisiin ratkaista rajatun alueen sisällä. Sovelluskehitystyöprosessissa onnistuminen edellyttää kuitenkin, että rakentamista tukeva tuotantosuunnitelma on selkeällä tavalla määritettyjä. [8] Riskinä kuitenkin on se, että mikäli ohjeistuksesta on luotu liian vaikeasti ymmärrettävää tai rajoitettua, käyttäjät tekevät sen suhteen hyvin todennäköisesti ohituksia. Sama koskee myös sovellussuunnittelussa käytettävää referenssiarkkitehtuuria: sen tulisi olla sovellussuunnittelijoiden ymmärrettävissä, jotta sen hyödyt konkretisoituisivat.

Dokumentaation ja kommunikaation voidaankin ajatella olevan tuoterunkoarkkitehtuurissa erittäin merkittävässä roolissa. [8] Mikäli käytetty dokumentaatio on laadultaan heikkoa tai vanhentunutta, ei sen avulla voida välittää kehittäjien kesken tarvittavaa ke-

hitystyötä tukevaa informaatiota. Dokumentaation avulla voidaan välittää myös tietoa tuoterunkoalustan yleisimmistä haasteista [28]. Erityisesti pienten ja keskisuurien yrityksien kohdalla, dokumentaatiota laiminlyöntiä voi tapahtua todennäköisemmin, sillä tuoterunkoalustan pystyttämiseen on suurempiin yrityksiin verrattuna tarjolla rajatumpi määrä kehitysresursseja. [19] Näin ollen resurssien käyttö painottuu pitkälti teknologiaan liittyvien ongelmien ratkaisemiseen, jolloin muut tuoterungon kannalta kuitenkin tärkeät asiat, kuten esimerkiksi dokumentaatio jäävät vähemmälle huomiolle.

Vaikka dokumentaatio olisi onnistuneesti toteutettu, sisältyy tuoterungon soveltamiseen silti muitakin riskejä. Esimerkiksi tuoteperheen piirteiden mallintamiseen sekä referenssiarkkitehtuuriin määrittämiseen tulee antaa erityistä huomiota [28]. Piirteiden mallintamisessa tärkeintä on tunnistaa tuoteperheessä oleellisesti muuttuvat ominaisuudet. Mikäli eri sovellusten välillä ei esiinny tarpeeksi eroavaisuuksia, voi tuoterunkoarkkitehtuuri olla lopulta täysin epäsovelias sekä kallis vaihtoehto. [8] Tunnistettu variaatio voi olla myös ylimitoitettua, jolloin tuoterungolla yritetään ratkaista liian geneerisiä ongelmia tai vaihtoehtoisesti variaatio voi vaatimuksiin nähden olla alimitoitettua. Tunnistettu variaatio vaikuttaakin suoraan referenssiarkkitehtuurin muotoon, ja mikäli se epäonnistuu, tuoterungon tarjoamaa tehokkuusnäkökulmaa ei saavuteta. Heikossa referenssiarkkitehtuurissa sovelluskomponentit eivät kytkeydy tai ole vuorovaikutuksessa asianmukaisesti, jolloin komponentteja ei saada tehokkaalla tavalla yhdisteltyä. [8] Alustan pohjalta luodut sovellustuotteet eivät saavuta tavoiteltuja laatuvaatimuksia eikä tuotevalikoimaan kuuluvia tuotteita ei saada rakennettua saatavilla olevista resursseista.

Variaation mallinnusta sekä referenssiarkkitehtuuria koskevaa riskiä voidaan madaltaa, suorittamalla arkkitehtuurille sen suunnitteluvaiheessa toiminnallisia ja ei toiminnallisia vaatimuksia tarkasteleva arviointi. Arkkitehtuurin arviointia varten on olemassa useita erilaisia menetelmiä, kuten SAAM, ATAM ja DCAR [29-31]. Arkkitehtuurin arvioinneissa voidaan tarkastella referenssiarkkitehtuurin kykyä täyttää pitkän tähtäimen tavoitteet ilman, että muut laatuvaatimukset kärsivät matkan varrella [20, s.221]. Tuoterunkoarkkitehtuurin tapauksessa arvioinnissa painopiste kohdistuu erityisesti referenssiarkkitehtuurin muunneltavuutta koskeviin laatuominaisuuksiin [11].

Onnistunut referenssiarkkitehtuuri mahdollistaa onnistuneen tuoterungon ja sitä kautta onnistuneen liiketoiminnan. Liiketoiminnallisiin näkökulmiin liittyvissä riskeissä merkittävämmäksi muodostuu tuoterukoarkkitehtuurin vaatima alkuinvestointi, jonka suuruus määräytyy valitun siirtymisstrategian pohjalta. Aikaisemmassa aliluvussa kuvatun proaktiivisen, eli *Big Bang* -lähestymistavan kuvailtiin sisältävän suurimman taloudellisen riskin, sillä valmiin resurssialustan luonti, ennen yhdenkään tuotteen julkaisua edellyttää suuren rahallisen investoinnin. Riskinä proaktiivisessa menetelmässä on kuitenkin se, että mikäli oleellisimmat vaatimukset muuttuvat kesken resurssien rakentamisen, voi tuotteen markkinoille pääsyssä syntyä viivettä. Riskiä on mahdollista pienentää käyttämällä siirtymisessä hyväksi inkrementaalisia lähestymistapoja [6]. Lisäksi riskiä voidaan punnita ennen käytännön soveltamista suorittamalla tuoterunkoalustalle alustava

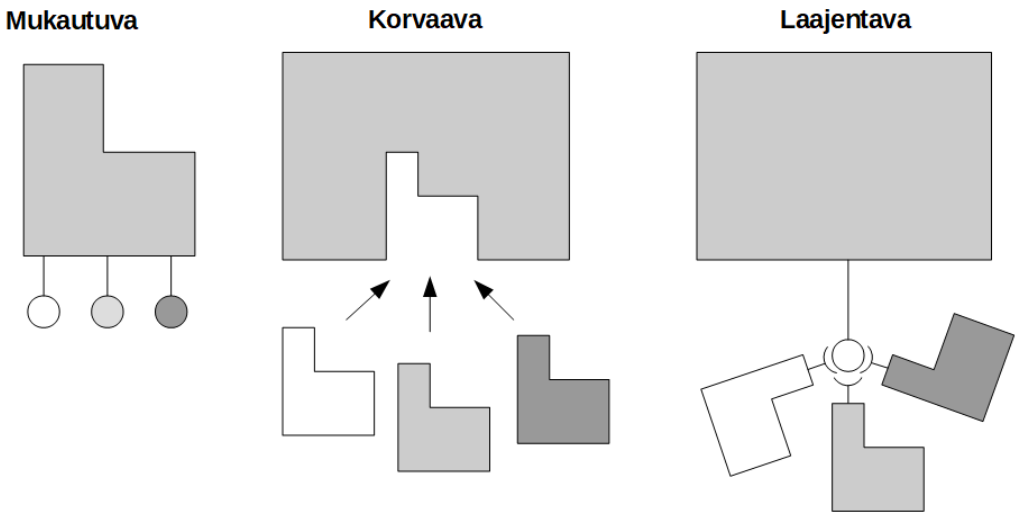
kustannusarvio käyttäen hyväksi aikaisemmin aliluvussa 2.2 esitettyjä tuoterunkoarkkitehtuurin laskentamalleja. Mallien käyttöön liittyy kuitenkin myös omat riskinsä. Mikäli laskennassa käytetty data on riittämätöntä tai sen muoto ei ole luotettavaa, syntyy laskentamalliin merkittävä virhemarginaali. Tällöin ei laskentamallien avulla voida luoda tarpeeksi vakuuttavia ennusteita tuoterunkoalustan taloudellisista vaikutuksista [8, 12, s.193].

3. MUUNNELTAVUTTA TUKEVAT MENETELMÄT JA TEKNOLOGIAT

Tässä luvussa esitetään tuoterungon muunneltavuutta edistäviä teknologioita sekä menetelmiä. Tarkastelussa ovat niin muunneltavuutta edistävät suunnittelukäytännöt, kuten myös puhtaat sovellustekniset menetelmät ja periaatteet.

Tuoterunkoarkkitehtuurin onnistuneesta soveltamisesta on yritysmaailmassa kertynyt vuosikymmenten aikana paljon positiivisia käytännön kokemuksia [32], jolloin myös erilaisia toimivia käytännön ratkaisuja muunneltavuuden realisoimiseen on tunnistettu. Piirremallin, tai minkä tahansa muun muunneltavuutta kuvaavan mallin realisointi käännetyksi koodiksi harvemmin onnistuu suoraan, vaikkakin keinoja siihenkin on myös tarjolla, joista esimerkkinä työkalu nimeltään *pure::variants* [33]. Teollisuuden sovelluskehityksessä, tuoterunkojen muunneltavuus mietitään kuitenkin usein omana suunnitteluongelmanaan, jossa hyödynnetään vakiintuneita ohjelmistokehityksen keinoja ja käytäntöjä [9].

Abstraktilla tasolla esitettynä, muunneltavuuden toteutustavat voidaan jakaa karkeasti kuvan 8 mukaisesti kolmeen eri kategoriaan, mukautuvaan (*adaptation*), korvaavaan (*replacement*) sekä laajentavaan (*extension*) [11, s.40].



Kuva 8. *Abstraktilla tasolla esiintyvät muunneltavuuden toteutustavat. Mukailtu lähteestä [11, s.41].*

Mukautuvalla tarkoitetaan sellaista ohjelmiston osaa, joka esiintyy yhtenä instanssina ja jonka käyttäytymistä voidaan rajapinnan kautta määritettävien parametrien avulla muuttaa. Korvaavassa menetelmässä tarjolla on useita erilaisia vaihtoehtomoduuleita, jolloin sovelluksen muunneltavuus realisoidaan valitsemalla sopivin moduuli. Laajentavassa tapauksessa sovellus tarjoaa ulos standardoidun rajapinnan, jonka kautta toiminnallisuutta pystytään lisäämään erilaisilla lisäosilla.

Käytännössä toteutettavaan tuoterunkoarkkitehtuurin muunneltavuuteen liittyen Gacek *et al.* sekä Tërnava *et al.* ovat julkaisuissaan koostaneet alla esitetyt tuoterunkoarkkitehtuurissa käytetyt muunneltavuusmenetelmät [34, 35].

- Koostaminen / delegointi
- Aspektipohjainen ohjelmointi
- Ehdollinen kääntäminen
- Dynaamisesti ladatut luokat (Java)
- Dynaamisesti linkitetyt kirjastot (DLL)
- Periyttäminen
- Rakentajien ylikuormitus
- Metodien ylikirjoitus
- Parametrisointi
- Staattiset kirjastot
- Kloonaus (*Copy & Paste*)
- Suunnittelumallit

Listatut menetelmät tarjoavat jokainen mahdollisuuden muunneltavuuden toteuttamiseen, mutta yhdenkään niistä ei voi sanoa olevan jokaiseen tilanteeseen täydellisesti sopiva ratkaisu [35]. Esimerkiksi kloonaus on arveluttava tapa toteuttaa muunneltavuus, mutta siitä huolimatta se on kuitenkin erittäin käytetty menetelmä [34]. Pienessä mittakaavassa yksinkertaisuutensa takia se voikin olla perusteltu ratkaisu, mutta, laajemmin käytettynä sitä soveltamalla voi syntyä ylläpidettävyysongelmia.

Menetelmän valintaan vaikuttaa myös variaation sitomishetki. Aliluvussa 2.4.3 esitettiin toteutustason jälkeen tapahtuvat muunneltavuuden sitomishetket, joita olivat sitominen kääntämisen, käynnistyksen tai ohjelman käynnin yhteydessä. Esimerkiksi perintään perustuvassa muunneltavuudessa muutos sidotaan käytännössä aina staattisesti ennen sovelluksen kääntämistä, kun taas delegointiin pohjautuvassa muunneltavuudessa sama toiminnallisuus on saavutettavissa ohjelman suorituksen aikaisesti [36]. Perinnässä etua saadaan kuitenkin siitä, sen toteuttama variaatio on läpinäkyvämpi paremman luettavuuden takia, sillä delegoinnissa variaatio sidotaan vasta sovelluksen käynnin aikana. Delegoinnin kautta saadaan kuitenkin myös etuja verrattuna perintään, kuten esimerkiksi riippumattomuus periytettäviin kantaluokkiin [36].

Muunneltavuusmenetelmien valinnassa on otettava huomioon myös epäsuoria tekijöitä, jotka vaikuttavat teknologian ulkopuolisiin seikkoihin. Näitä ovat esimerkiksi menetelmän käyttöönottamisen vaatima tietotaito ja tätä kautta soveltamisen kohderyhmä, menetelmän hinta, niin toteuttamisessa kuin myös sen menetelmän harjoittelemisessa sekä menetelmän vaikutus muihin laatutekijöihin, kuten vaikka suorituskykyyn [37]. Esimerkiksi kohderyhmään liittyen, muunneltavuutta tukevien suunnittelumallien oikeaoppinen soveltaminen vaatii muunneltavuuden soveltajalta tietämystä olio-ohjelmoinnista, kun taas esimerkiksi parametreihin perustuvassa menetelmässä riittää, että soveltaja ymmärtää vain parametrien tarkoitusperän.

3.1 Modulaarinen ohjelmistokehitys

Modulaarisuus on yleinen konsepti kompleksisuuden hallintaan. Jakamalla järjestelmäkokonaisuus useaan pienempään itsenäiseen alijärjestelmään, saadaan aikaiseksi helpommin ymmärrettäviä sekä ylläpidettäviä kokonaisuuksia. [38] Modulaarisuus mahdollistaa kokonaisuuksien kehittämisen tavoin, joissa yksittäiset tiimit tai kehittäjät voivat itsenäisesti pureutua yhden vastuualueen ongelmien ratkaisemiseen. Modulaarinen kehitystyö koostuu [38] mukaan kolmesta eri osasta:

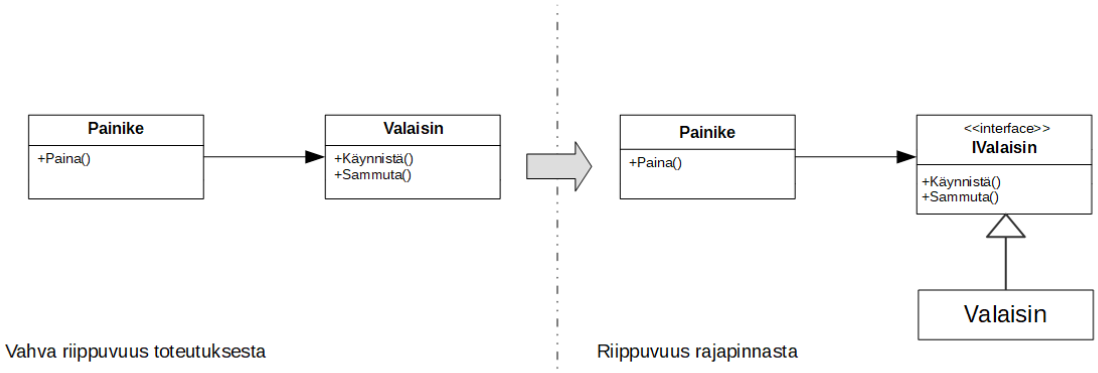
- Arkkitehtuurin määrittely, jossa eri moduulien vastuualueet määritellään.
- Rajapinnat, jotka määrittelevät eri moduulien interaktion.
- Standardit, jotka peilaavat moduulien yhdenmukaisuuden suunnittelusäännöstöön.

SoC, eli *Separtion of concerns* on ohjelmistokehityksessä syntynyt suunnitteluperiaate, jonka mukaan tietyt vastuualueet tulisi sovelluksen rakenteessa eristää niiden loogisen toiminnallisuuden mukaan omiksi kokonaisuuksiksi [39]. Modulaarisessa arkkitehtuurissa vastuualueiden tulisikin olla tällä tavalla määritettyjä, jotta yksittäiset ominaisuudet olisivat testattavissa sekä kehitettävissä riippumatta muiden kokonaisuuksien olemassa olosta [40]. Onnistunut modulaarinen arkkitehtuuri mahdollistaa helpomman tavan tehdä muutoksia, päivityksiä sekä testejä ilman koko järjestelmää läpileikkaavaa muutostyötä, jolloin esimerkiksi uusien teknologioiden käyttöönottaminen voi tapahtua nopeammin [41].

Toinen merkittävä modulaarisuutta edistävä suunnitteluperiaate on **DRY,** eli *Dont Repeat Yourself*, joka edellyttää, että jokaisen järjestelmässä olevan tiedon tulisi esiintyä järjestelmässä yksittäin sekä yksiselitteisenä [42]. Näin ollen toiminnallisuuden monistamista ei tulisi tehdä kopio & liitä tekniikkaa hyväksi käyttäen vaan käyttäen abstrakteja esityksiä [43]. Proseduraalisessa ohjelmointityylissä yksittäisenä abstraktina yksikkönä voidaan nähdä proseduuri, eli funktio. Olio-ohjelmoinnissa samanlaisen modulaarisen yksikön muodostaa luokka, joka koostuu metodeista, sekä attribuuteista. Komponenttipohjaisessa ohjelmistokehityksessä modulaarisen abstraktin yksikön muodostaa komponentti, joka tarjoaa yleensä luokkaan nähden suurempaa kokonaisuutta palvele-

vaa toiminnallisuutta. Toistuvuuden eliminointi parantaa arkkitehtuurin kokonaisvaltaista rakennetta ja edesauttaa näin ollen sovelluksen ylläpidettävyyttä sekä muunneltavuutta, sillä kloonaukseen verrattuna abstraktioiden avulla tiettyyn ominaisuuteen kohdistuva muutos tarkoittaa luonnollisesti pienempää työmäärää.

Vastuualueiden ja näiden kautta moduulien hahmottamisen jälkeen eri moduulien välille tulisi määritellä rajapinnat. Olio-ohjelmoinnin kontekstiin liittyen Robert C. Martin on esittänyt viisi suunnitteluperiaatetta (SOLID), joiden tarkoituksena on ohjeistaa ymmärrettävämmän, joustavamman sekä ylläpidettävämmän ohjelmakoodin tuottamista [44]. Näistä suunnitteluperiaatteista viimeisin, eli ***Dependency Inversion Principle***, suomeksi riippuvuuksien kääntö on suunnitteluperiaate, jota voidaan modulaarista kehitystyötä ajatellen pitää merkittävänä. Ideana on, että korkeamman tason moduulien ei tulisi olla vahvasti riippuvaisia alemman tason moduuleiden toteutuksista. Sen sijaan riippuvuuksien tulisi kohdistua toteutuksien sijaan rajapintoihin. Kuvassa 9 on havainnollistettu ajatusta: sen sijaan, että kuvassa esiintyvä painike-luokka olisi suoraan riippuvainen valaisimen toteutuksesta, esiintyy riippuvuus valaisimen toteuttamaan rajapintaan *IValaisin*. Rajapinta toimii näin ollen sopimuksena kahden luokan välillä, eikä se ota kantaa, millä tavoin sen taustalla tapahtuva toteutus on tehty. Rajapinta kuitenkin määrittelee vuorovaikutukselle säännöt.



Kuva 9. *Esimerkkitoteutus DIP-periaatteen soveltamisesta. Mukailtu lähteestä [45, luku 11].*

Proseduraalisessa ohjelmointitavassa vahvojen riippuvuusketjujen luominen on yleensä väistämätöntä. [45, luku 11] Sen sijaan olio-ohjelmoinnissa näistä suorista riippuvuussuhteista voidaan päästä eroon soveltamalla DIP-suunnitteluperiaatetta. Menetelmän soveltaminen ei kuitenkaan ole välttämättä kaikissa tilanteissa tarpeellista, sillä jossain vaiheessa sovellusta vahvan riippuvuuden tekeminen on välttämätöntä. [40] Esimerkiksi mikäli kysymyksessä luokka, joka on rakenteeltaan vakaa, jolloin siihen ei kohdistu käytännössä koskaan muutoksia, voi riippuvuus olla suoraan toteutukseen. Tästä esimerkkinä on monella eri ohjelmointikielellä esiintyvä merkkijonoja käsittelevä STRING-luokka [40, luku 11].

Hyväksi todettujen suunnitteluperiaatteiden noudattaminen on monessa suhteessa tärkeää, mikäli tuoterungon resurssialustan arkkitehtuurista halutaan laatuominaisuuksiltaan muunneltava sekä helposti ylläpidettävä. [21, s.66] Lisäksi toiminnallisuuksien eristämistä loogisesti sekä löyhien kytköksien muodostaminen on tärkeää kaikilla sovelluksen arkkitehtuurin rakennekokonaisuuksilla. Tyypillistä on kuitenkin se, että mikäli ominaisuudet tehdään kiireessä, annettaan ylläpidettäville ja modulaarisille toteutusratkaisuille liian vähän painoarvoa [40]. Tällöin ominaisuudet tehdään tavalla, joka pitkässä juoksussa johtaa arkkitehtuurin rapautumiseen.

3.2 Suunnittelumallit

Oliopohjaisten suunnittelumallien hyödyntäminen on klassinen menetelmä skaalautuvan arkkitehtuurin aikaansaamiseksi. Suunnittelumallit, samalla tavoin kuin suunnitteluperiaatteet, eivät varsinaisesti ole keksittyjä, vaan ne ovat ajan kuluessa kypsyneitä tapoja toteuttaa ratkaisuja erilaisiin toistuviin suunnitteluongelmiin [36]. Jotta voidaan puhua niin sanotusta yleisestä suunnittelumallista, tulee ongelman toistua tarpeeksi usein. [2, s.85] Ratkaisun tulee myös olla yleistettävissä laajalle sovellusalueelle. Mikäli malli esittää ratkaisun koskien vain yhtä sovellusaluetta, on kyseessä todennäköisimmin tätä kyseistä sovellusta koskeva yksittäinen analyysimalli.

Suunnittelun tehtävänä on yleisesti optimoida suunnitelma jonkin suunnittelukriteerin funktiona, joka yleensä tapahtuu muiden kriteerien kustannuksella. [2, s.34] Tämä johtuu siitä, että suunnittelutyö tehdään yleensä rajattujen resurssien sisällä. Kriteerit voivat olla esimerkiksi seuraavanlaisia [2, s.34]:

- Suorituskyky
    - Pahimmassa tapauksessa
    - Keskimääräinen
- Ennustettavuus
- Aikataulutus
- Resurssien minimointi
    - Muisti
    - Lämpötila
    - Paino
- Uudelleenkäytettävyys
- Siirrettävyys
- Ylläpidettävyys
- Luettavuus
- Laajennettavuus
- Kehitykseen käytettävä aika/vaiva
- Turvallisuus
- Luotettavuus

- Tietoturvallisuus

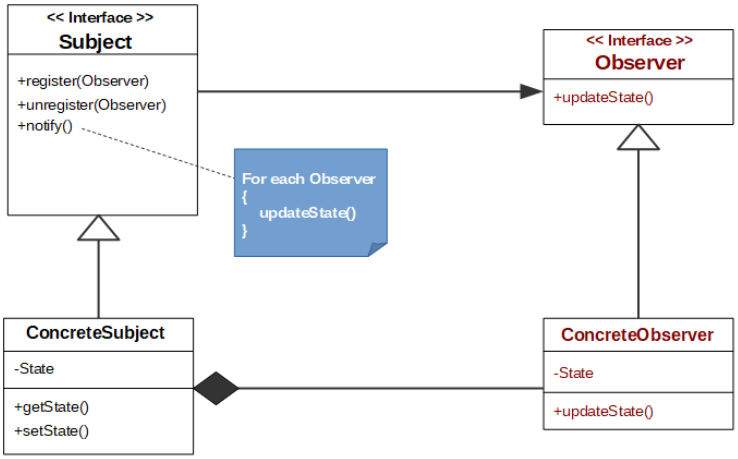
Tuoterunkoarkkitehtuuria ajatellen tärkeitä kriteereitä ovat uudelleenkäyttö, skaalautuvuus sekä ylläpidettävyys. Näiden painottaminen suunnittelussa saattaa kuitenkin joissain tapauksissa vaikuttaa negatiivisesti esimerkiksi sovelluksen suorituskykyyn [20]. Siksi tärkeää onkin löytää sovellustapaukseen sopiva optimaalinen ratkaisu priorisoimalla laatuvaatimukset tärkeytensä mukaan [2, s.34].

Tässä aliluvussa käydään läpi muutamia suunnittelumalleja, joiden avulla tuoterunkoarkkitehtuurin vaatima muunneltavuus on mahdollista saavuttaa. Muunneltavuuden mahdollistavia sunnittelumalleja on julkaisuissa sekä kirjallisuudessa tunnistettu seuraavanlaisia: *observer*, *template-method*, *strategy*, *decorator*, *visitor* ja *composite* [9, 34, 46, 47]. Tässä aliluvussa yksityiskohtaisempaan tarkasteluun otetaan *observer*, *template-method*, *strategy*, *decorator* sekä *composite*. Näiden lisäksi mukaan tarkasteluun on otettu modulaarista arkkitehtuuria edistävä *dependency injection*, eli riippuvuuksien syöttö-suunnittelumalli, joka mahdollistaa DIP-suunnitteluperiaatetta noudattavan tavan toteuttaa luokkien välille löyhät riippuvuudet [48].

Joustavaa muunneltavuutta edistävät suunnittelumallit tarjoavat mahdollisuuksia ajon aikaiseen muunneltavuuteen. Vaihtoehtoisesti muunneltavuus voidaan aktivoida ohjelman käynnistyksen aikana, jolloin variaatio realisoituu esimerkiksi määritettyjen alustusparametrien mukaisesti.

Observer, eli tarkkailija-suunnittelumalli mahdollistaa löyhien kytköksien muodostamisen eri kokonaisuuksien välille [49]. Tarkkailija-suunnittelumallin avulla objektit voivat olla keskinäisessä vuorovaikutuksessa ilman suoraa riippuvuussuhdetta. Kyseessä tapahtumalähteen ja kuuntelijoiden osalta 1:n –vuorovaikutuksesta [36].

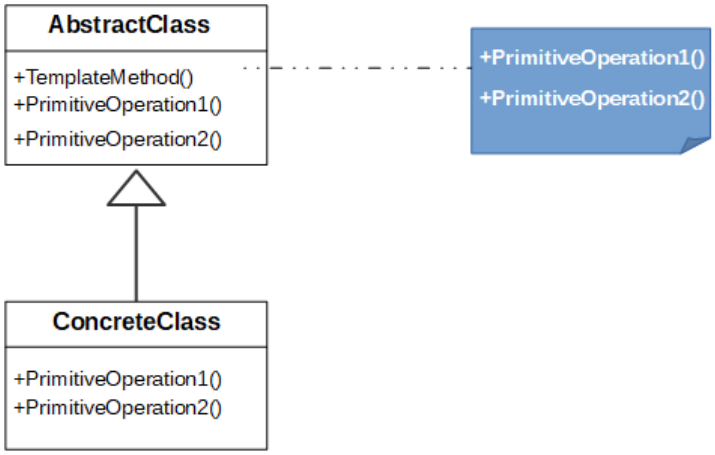
Suunnittelumallin UML-luokkakaavio on esitetty kuvassa 10. Mallin ajatuksena on se, että tarkkailijat rekisteröityvät kuuntelemaan tapahtumanlähdettä. Siinä vaiheessa, kun tapahtumalähteessä ilmenee muutoksia, lähetetään tarkkailijoille uusi tieto. Suunnittelumalli mahdollistaa parannuksen suoritettavan ohjelman suorituskykyyn siltä osin, että sen avulla päästään eroon jatkuvasta pollauksesta, sillä tieto siirtyy asynkronisesti [20, s.88].



Kuva 10. *Tarkkailija-suunnittelumallin UML-luokkakaavio. Mukailtu lähteestä [36].*

Tuoterunkoarkkitehtuuria ajatellen Tarkkailija-suunnittelumalli mahdollistaa esimerkiksi eri piirteitä/ominaisuuksia toteuttavien ohjelmakomponenttien välille löyhät kytkökset, jolloin erilaisia ominaisuuksia toteuttavien komponenttien vaihtaminen onnistuu tarvittaessa joustavammin. [9, luku 4.2]. Tällöin riittää, että kuuntelijaksi rekisteröityvät vaihtoehtoiset ominaisuuskomponentit toteuttavat tarvittavan tarkkailija-rajapinnan.

Template method, suomeksi kehysmetodi [49] on perintään perustuva tapa toteuttaa algoritmin variaatio [36]. Suunnittelumallin avulla yleisesti käytettävä metodin runko saadaan abstrahoitua kantaluokkaan. Kantaluokan periyttävä aliluokka pystyy tarvittaessa ylikirjoittamaan kantaluokan metodin toteutuksen, jolloin algoritmi saadaan vaihdettua. Toiminta on esitetty kuvassa 11 UML-luokkakaavion muodossa.

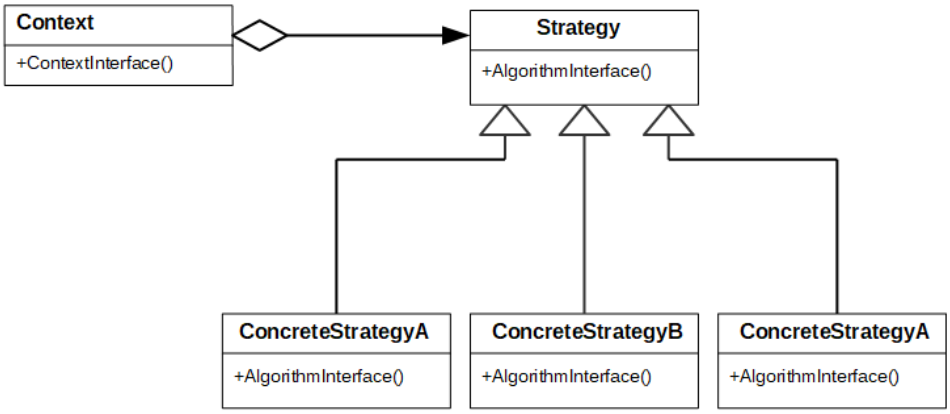


Kuva 11. *Kehysmetodi-suunnittelumallin UML-luokkakaavio. Mukailtu lähteestä [36].*

Suunnittelumallin heikkoutena on kuitenkin perinnästä aiheutuva aliluokan vahva sidos sen kantaluokkaan, joka esimerkiksi rikkoo suoraan SOLID-suunnitteluperiaatteen mukaista DIP–menetelmää [45]. Lisäksi perintään perustuvassa muunneltavuudessa kysymyksessä on staattisesti käännösvaiheessa tapahtuvasta variaatiosta, jonka vuoksi muunneltavuuteen ei pystytä enää kääntämisen jälkeen vaikuttamaan.

Tuoterunkoa ajatellen Kehysmetodi tarjoaa silti mahdollisuuden toteuttaa esimerkiksi toimintaa suorittaville algoritmeille variaatio. [9, luku 4.2] Tällöin variaatio aktivoidaan niin, että kantaluokan perivä aliluokka yliajaa algoritmia suorittavat metodit.

Strategy-suunnittelumalli, suomeksi Strategia [49] mahdollistaa kehysmetodin tavoin, algoritmia koskevan variaation [36]. Kehysmetodiin verrattuna Strategia-suunnittelumallissa variaatio toteutuu perinnän sijaan delegoinnin avulla. Kyseessä on 1:n vaihtoehdon vaihtaminen. [36]



Kuva 12. *Strategia-suunnittelumallin UML-luokkakaavio. Mukailtu lähteestä [36].*

Kuva 12 esittää strategia –suunnittelumallin UML-luokkakaavion. Strategia-suunnittelumallissa eri algoritmeja toteuttavat strategialuokat jakavat yhteisen *Strategy*-rajapinnan. *Context*-luokka koostuu *Strategy*-rajapinnan sisältävästä objektista ja alustamalla sen tilalle eri algoritmin, mutta saman rajapinnan toteuttava strategia-objekti, saadaan algoritmin toteutus tarvittaessa vaihdettua. Kuvassa 13 on pseudokoodin avulla havainnollistettu esimerkki strategian vaihdosta.

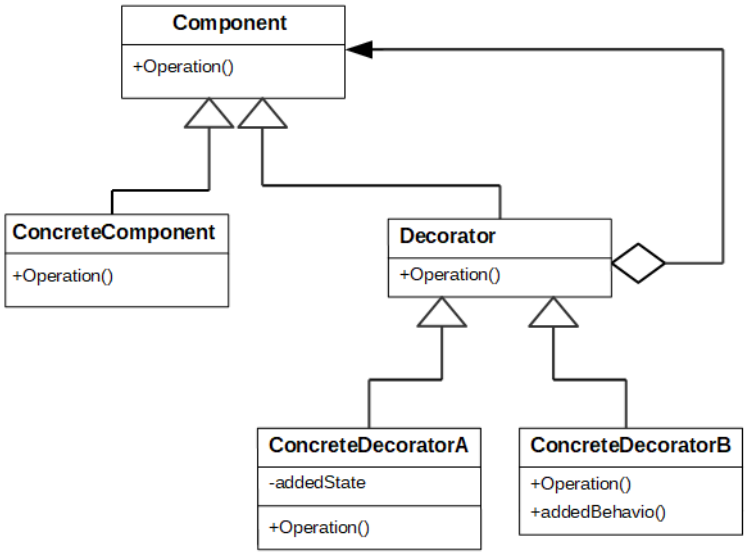
```
//Alustetaan aluksi käyttöön strategia A:
context := new Context(new concreteStrategyA());

//Strategian vaihto A -> B
context := new Context(new concreteStrategyB());
```

Kuva 13. Alustamalla instanssi uudelleen, saadaan strategia vaihdettua

Muunneltavuus tapahtuu delegointiin pohjautuen, jonka vuoksi se on mahdollista tehdä sovelluksen kääntämisen jälkeenkin. Tuoterunkoarkkitehtuurissa strategia-

suunnittelumallia voidaan käyttää sovelluksen suorituksen aikaisena vaihtoehtojen valitsijana. Vaihtuva kokonaisuus voi esimerkiksi olla sovelluskomponentti, joka suorittaa tietyn rajapinnan toiminnallisuuden valinnasta riippuen eri tavalla [12, s.225].

Decorator, suomeksi Koristelija [49] on Strategia-suunnittelumallin tavoin delegointiin perustuva suunnittelumalli, joka mahdollistaa objektin toiminnallisuuden laajentamisen dynaamisesti [36]. Sen sijaan, että luokan toiminnallisuutta laajennettaisiin esimerkiksi periyttämällä, voidaan lisätoiminnallisuutta kääriä luokan ympärille erilaisten koristelija -objektien avulla.



Kuva 14. *Koristelija -suunnittelumallin UML-luokkakaavio. Mukailtu lähteestä [36].*

Kuvassa 14 on esitetty Koristelija suunnittelumallin rakenne UML-luokkakaavion muodossa. Toiminta perustuu siihen, että *Decorator*-luokka sisältää *Component*-luokan instanssin samalla perien *Component*-luokan rajapinnan. Konkreettiset koristelijat, eli *ConcreteDecorator*-luokat perivät sittemmin *Decorator*-rajapinnan. Komponentin toimintaa voidaan laajentaa luomalla ilmentymät halutuista *ConcreteDecorator*-luokista ja alustamalla laajennusta kaipaava komponentti näiden rakentajan läpi, joka sittemmin palauttaa komponentin, jonka ympärille on käärytty lisätoiminnallisuus (kuva 15). Koristellun luokan rajapinta pysyy kuitenkin alkuperäisenä, eli lisätoiminnallisuus toteutuu aina objektin jo olemassa olevan rajapinnan yhteyteen.

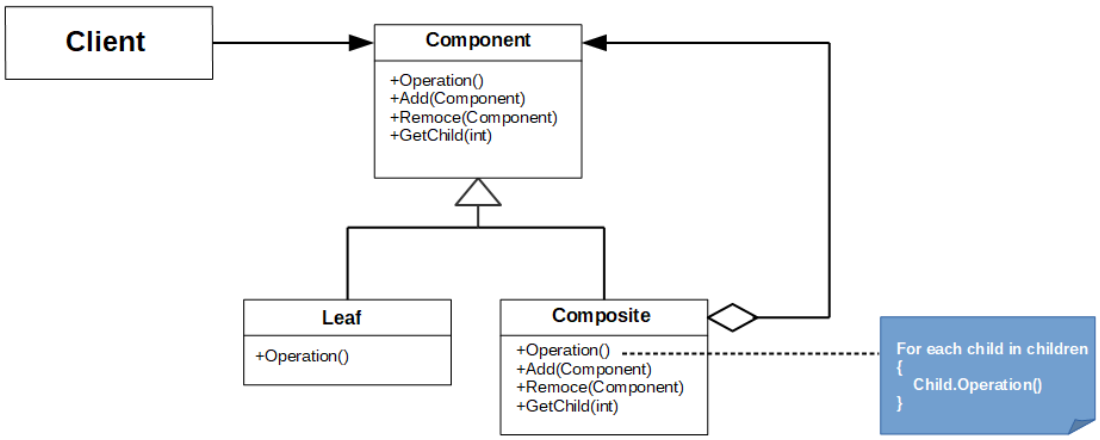
```
//Komponentin toiminnallisuutta laajennetaan alustamalla
//se uudelleen koristelijan avulla
component := new concreteDecoratorA(component);
```

Kuva 15. *Koristelija-ominaisuuden kääriminen Component-luokan instanssiin*

Tuoterunkoa ajatellen suunnittelumalli mahdollistaa esimerkiksi keinon kääriä yksinkertaisen kantaluokan ympärille vaihtoehtoista toiminnallisuutta tarvittaessa ohjelman suo-

rituksen aikaisesti. [9, luku 4.2] Koristelijoiden määrää ei ole rajattu, jolloin objektin toiminnallisuutta voidaan tarvittaessa laajentaa mielivaltaisesti.

Composite, suomeksi rekursiokooste suunnittelumalli mahdollistaa objektien koostamisen puumaiseen hierarkiseen rakenteeseen [20, 36]. Suunnittelumallin avulla joukko objekteja voidaan koostaa yhteen niin, että ne yhdessä muodostavat käsiteltävän kokonaisuuden.



Kuva 16. *Rekursiokooste-suunnittelumallin UML-luokkakaavio. Mukailtu lähteestä [36].*

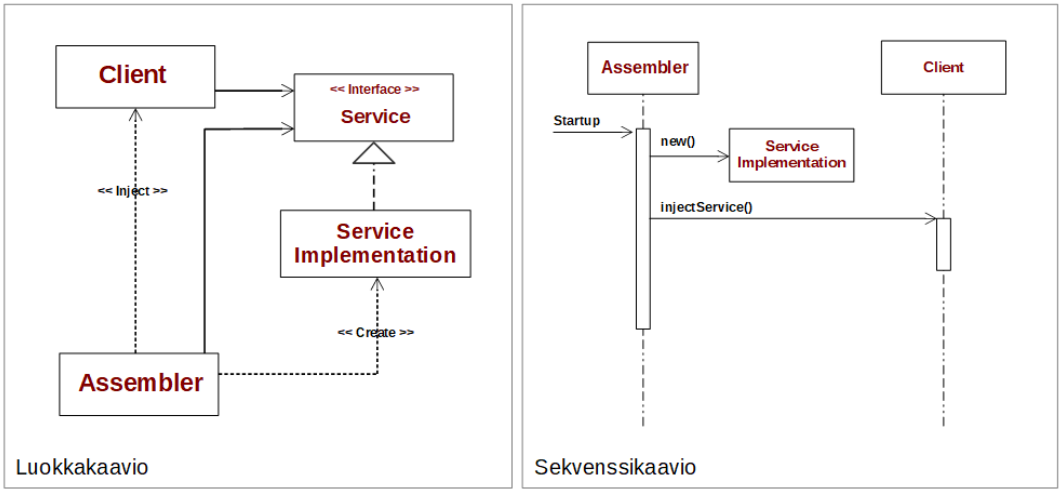
Kuvassa 16 on esitetty suunnittelumallin UML-luokkakaavio. Ajatuksena on, että juurielementti *Composite* sekä primitiivisempi lapsielementti *Leaf* jakavat saman yhteisen rajapinnan *Component*, jonka kautta kokonaisuutta voidaan käsitellä. Juurielementin alle on mahdollista sisällyttää lapsielementtejä tai muita juurielementtejä. Tällä tavoin on mahdollista saada aikaiseksi puumainen rakenne. Kokonaisuutta käyttävälle asiakkaalle (*Client*) rakenne näkyy yhtenä abstraktiona [20, s.114].

Tuoterunkoarkkitehtuurissa rekursiokooste-suunnittelumallia voidaan hyödyntää rakentamalla sellaisia oliokokonaisuuksia, jotka mukautuvat varioituvien ominaisuuksien suhteen. Esimerkki tällaisesta voisi olla graafisten elementtien dynaaminen kokoaminen sovelluksen tukemien ominaisuuksien perusteella.

Dependency Injection eli riippuvuuksien syöttö on menetelmä, joka mahdollistaa *Dependency Inversion Principle* -menetelmän mukaisesti luokkien välille riippumattomuuden [48]. Termin *dependency injection* toi ilmi Martin Fowler vuonna 2004 kirjoittamassaan artikkelissaan [50].

Kuvassa 17 on esitetty esimerkki suunnittelumallin toiminnasta UML-luokkakaavion sekä sekvenssikaavion avulla. *Client*-luokan riippumattomuus *Service*-luokan toteutukseen saavutetaan niin, että *Client*-luokan riippuvuus kohdistetaan *Service*-luokan toteutuksen sijaan *Service*-luokan toteuttamaan rajapintaan. *Assembler*-luokan tehtävänä on

luoda *Service*-luokan instanssi ja syöttää se *Client*-olion käytettäväksi. Näin ollen *Client*-olion ei tarvitse itse luoda *Service*-luokan ilmentymää, jolloin *Client* sekä *Service* -luokkien toteutuksien välille saadaan toisistaan riippumaton suhde.



Kuva 17. *Riippuvuuksien syöttöä kuvaava UML-luokka- ja sekvenssikaavio. Mukailtu lähteestä [51].*

Tyypillisesti instanssi syötetään sitä tarvitsevalle *Client*-luokalle rakentajan kautta parametrina. Muita tyylejä ovat erillinen syöttöön tarkoitettu *Set*-metodi sekä rajapinta-injektio [50].

Riippuvuuksien syöttö voidaan hoitaa tätä tarkoitusta varten räätälöidyllä kehyksellä, joista usein käytetään myös termiä *IoC-container*, eli *Inversion Of Control*-säiliö [50]. Esimerkkejä tällaisia säiliöitä tarjoavista alustoista ovat Java Spring sovelluskehys [52] sekä .NET komponenttikirjaston kanssa yhteensopiva Unity Container [53]. IoC-säiliöissä voidaan erilaisten määrittelytiedostojen tai skriptien avulla vaikuttaa siihen, minkälainen konkreettinen luokka sidotaan mihinkin rajapintaan. Tällöin ne myös mahdollistavat keinon määritellä objektien variaatiota ilman sovelluksen uudelleenkääntöä.

Riippuvuuksien syöttö -suunnittelumallin ensisijaisena tarkoituksena on parantaa luokkien testattavuutta [48], mutta samalla se myös mahdollistaa tuoterunkoarkkitehtuuria edistävän tavan toteuttaa modulaarinen ohjelmistoarkkitehtuuri. Riippuvuussuhde ei näin ollen esiinny suoraan luokkien toteutuksiin vaan palveluita tarjoaviin rajapintoihin. Rajapinnan säilyessä luokan toteutus on mahdollista vaihtaa tarvittaessa joustavammin. Tämä parantaa luokkien uudelleenkäyttöä, sillä yksittäisen luokan ei tarvitse kantaa harteillaan pitkää objektien muodostamaa riippuvuusketjua [54].

Suunnittelumallit tarjoavat hyväksi havaittuja keinoja yleisiin ohjelman rakennetta koskeviin suunnitteluongelmiin ja tätä kautta tuoterunkoarkkitehtuurin edellyttämään muunneltavuuteen. Suunnittelumallien hyvänä puolena on myös niiden riippumattomuus toteutuskielestä, riittää vain, että ohjelmointikieli tukee mallin edellyttämiä paradigmoja, kuten esimerkiksi olio-ohjelmointia. Suunnittelumallien onnistunut soveltami-

nen vaatii kuitenkin, että niiden olemassaolo dokumentoidaan hyvin. Muuten vaarana on niiden häviäminen koodin sekaan [20, s.118].

Kuten kaikilla menetelmillä, myös suunnittelumalleilla on omat heikkoutensa: suunnittelumallit eivät tarjoa valmista koodipohjaa toteutuksen realisointiin, vaan pelkästään abstraktin kuvauksen. Lisäksi, mikäli mallin tarjoaman ratkaisun tulkitaan väärin tai mikäli mallia sovelletaan väärässä tilanteessa saattaa se aiheuttaa turhaa monimutkaisuutta sovelluksen arkkitehtuuriin ja lopulta itse sovelluksen lähdekoodiin. Lisäksi suunnittelumallien aiheuttama koodillinen ja arkkitehtuurinen kuorma saattaa joissain tapauksissa vaikuttaa negatiivisesti myös sovelluksen suorituskykyyn, joka saattaa nousta kynnyskysymykseksi esimerkiksi sulautetuissa järjestelmissä [9, 20].

3.3 Arkkitehtuurimallit

Arkkitehtuurimalli määrittelee sovelluksen korkean tason rakenteen sekä eri rakenteiden väliset riippuvuussuhteet. [20] Siinä missä suunnittelumallit ottavat kantaa tiettyjen ongelmien ratkaisuun yksityiskohtaisemmin, arkkitehtuurimallit ottavat kantaan isomman kokonaisuuden rakenteeseen. Usein arkkitehtuurimallit auttavat hahmottamaan eri ohjelmamoduulien vastuualueita näiden tyylien perusteella.

Tuoterunkoarkkitehtuuria edellyttävä modulaarisuus asettaa vaatimuksia myös arkkitehtuurimallille. Mallin tulisi tarjota edellytyksiä erilaisten vaihtoehtojen toteuttamiselle ja sen tulisi mahdollistaa vastuualueiden ja ominaisuuksien selkeän jakamisen, pitäen eri osien keskinäiset riippuvuudet samalla mahdollisimman löyhinä. Lisäksi arkkitehtuurimallin tulisi ottaa huomioon tuoterungossa *ajan suhteen* tapahtuvat muutokset, jolloin arkkitehtuurin rakenne mahdollistaisi sellaisten kokonaisuuksien eristämisen, joita tarvitsee muita useammin muuttaa.

Kerrosarkkitehtuuri mahdollistaa ohjelmiston vastuualueiden jakamisen kerroksiin horisontaalisesti niin, että ylemmät kerrokset käyttävät hyväkseen alemman kerroksen tarjoamia palveluita [20]. Tyypillisesti tavoitteena on luoda tiettyyn vastuualueisiin keskittyviä kerroksia, jolloin yksittäisen kerroksen kehittyminen voi tapahtua vaakasuunnassa ilman, että koko sovelluksen toiminnallisuutta tarvitsee muuttaa [20]. Esimerkiksi käyttöliittymän eriyttäminen omaksi kerrokseksi sovelluslogiikasta on yleinen sekä suositeltava tapa soveltaa kerrosarkkitehtuuria [21, s.186].

Tuoterunkoarkkitehtuuria ajatellen se mahdollistaa paremman uudelleenkäytettävyyden, sillä vaihtamalla tietty kerros toiseen voidaan saada sovellus toimimaan esimerkiksi uudessa ympäristössä [20]. Kerrosarkkitehtuurin yleisin mahdollinen ongelma on sen negatiivinen vaikutus sovelluksen suorituskykyyn. [2, 20] Tarvittaessa suorituskykyä voidaan optimoida tekemällä ohituksia eri kerrosten väleillä.

Viestinvälitysarkkitehtuuri (Broker-pattern, Message passing) mahdollistaa vuorovaikutuksen hajautettujen komponenttien välillä [21, s.237]. Keskenään kommunikoivien

komponenttien ei tarvitse olla suoraan tietoisia toisistaan, sillä kommunikaatio tapahtuu keskitetyn viestinvälittäjän kautta. [20, s.139] Komponenttien ei näin ollen tarvitse toteuttaa mitään yksittäisiä staattisia rajapintoja vaan ainoastaan viestien välittämiseen tarkoitettu rajapinta. Viestinvälittäjä tarjoaa rajapinnan palveluiden rekisteröimiselle, jolloin komponentit voivat liittyä dynaamisesti osaksi viestinvälitystä [21, 55]. Komponenttien välinen viestien välitys voi tapahtua tapahtumapohjaisesti (Tarkkailija-suunnittelumalli), jolloin komponenttien väliset viestit siirtyvät asynkronisesti [21, s.239].

Viestinvälitysarkkitehtuurin avulla komponenttien välisestä viestinnästä saadaan astetta geneerisempää, jolloin se soveltuu varsinkin sellaisen tuoterunkoarkkitehtuurin tarpeisiin, jossa komponenttien rajapintoja ei pystytä vielä täysin ennustamaan. Lisäksi viestinvälitysarkkitehtuuri mahdollistaa hajautetun luonteensa vuoksi sovelluskielestä riippumattoman kommunikoinnin, jolloin ominaisuuksia tarjoavia komponentteja voidaan tarvittaessa toteuttaa sekaisin eri ohjelmointikielillä.

Palvelupohjaisessa arkkitehtuurissa (*Service Oriented Architecture*) sovellus koostuu hajautetuista sekä itsenäisistä web-palvelukomponenteista, jotka voivat fyysisesti sijaita sovelluksen ulkopuolisille palvelimilla [56]. Toimiva sovellus toteutetaan tällöin koostamalla yhteen tarvittavat palvelut sekä suorittamalla niitä halutussa järjestyksessä [8]. Palvelupohjaisessa arkkitehtuurissa komponenttien välinen viestitys tapahtuu standardoitujen web-palveluteknologioiden, kuten esimerkiksi SOAP-viestiprotokollan tai tilattoman REST-arkkitehtuurin mukaisesti, jolloin se on teknologialtaan alustariippumaton. Tuotekohtainen varioitavuus saavutetaan palvelupohjaisessa arkkitehtuurissa vaihtamalla palvelukomponentteja halutun variaation mukaisesti [8].

Mikropalveluarkkitehtuurilla tarkoitetaan arkkitehtuurimallia, jossa sovellus koostetaan yksittäisistä, omissa prosesseissa suoritettavista palvelukomponenteista, jotka julkaistaan, skaalataan sekä testataan omina yksiköinään ja joilta jokaiselta löytyy yksi vastuualue, jonka puolesta se tarjoaa palveluitaan [41, 57]. Mikropalveluarkkitehtuuri eroaa perinteisestä monoliittisesta palvelupohjaisesta arkkitehtuurista siltä osin, että mikropalveluiden oletetaan olevan kooltansa hienojakoisempia ja kevyitä ja niiden ajatellaan sijoittuvan omiin suoritettaviin prosesseihin. Mikropalvelut ovat siis yksi arkkitehtuurillinen malli toteuttaa modulaarinen tuoterunkoarkkitehtuuri.

3.4 Parametrisointi

Parametrisointi on yksinkertainen, nopea sekä lähes kaikissa ohjelmointiympäristöissä mahdollinen tapa toteuttaa tuoterunkoarkkitehtuurin muunneltavuus. [9, luku 4.1] Yksinkertaisimmillaan parametrisointiin pohjautuva muunneltavuus saadaan aikaiseksi *if..else* tai *switch case* –tyyppisiä ehtoja käyttämällä.

Parametrit voidaan kovakoodata sovellukseen staattisesti tai sitten ne voidaan dynaamisesti lukea esimerkiksi käynnistyksen yhteydessä erillisestä konfigurointitiedostosta. Parametrit voiva sijaita, esimerkiksi tietokannan tauluissa, jolloin ohjelman kontrollointiin tarkoitetut muuttuja sekä sääntöihin vaikuttavat parametrit on eriytetty toisistaan [58, 59].

Parametrit voivat siirtyä sovelluksessa ketjumaisesti metodilta metodille tai sitten niiden arvot voidaan lukea ympäri sovellusta esimerkiksi globaaleiden muuttujien avulla. Useassa moduulissa läpileikkaavat globaalit muuttujat aiheuttavat kuitenkin sovelluksessa moduulien keskinäisiin riippuvuuksiin vahvoja sidoksia. Tällöin parempi menetelmä parametrien siirtämiseen on parametriobjektin käyttö, jossa tarvittavat parametrit koostetaan objektin sisälle. [60] Tällöin parametreille saadaan myös vahvempi tyypitys. Objektia voi sittemmin siirtää helposti ympäri sovellusta siellä missä mahdollisesti parametreja tarvitaan.

Ehtolauseisiin pohjautuva parametrisointi synnyttää helposti sovelluksen rakenteeseen negatiivisia vaikutuksia, sillä ehtolauseiden avulla toteutettu muunneltavuus on usein ad-hoc tyyppinen. [9, luku 4.1] Esimerkiksi, mikäli jotain ominaisuutta ei koskaan tarvita, on se tästä huolimatta aina osa sovelluksen lähdekoodia. Tämä saattaa aiheuttaa ongelmia turvallisuuteen tai suorituskykyyn liittyen. Lisäksi ehtolauseiden suuri määrä kasvattaa sovelluksen kompleksisuusastetta, jolloin lähdekoodin ymmärtäminen vaikeutuu [61].

Myös käytettyjen parametrien määrä vaikuttaa tuoterunkoarkkitehtuurin muotoon. [8] Liiallinen parametrimäärä saattaa aiheuttaa sen, että parametrien muodostamaa kokonaiskuvaa on hyvin vaikea hahmottaa, kun taas liian vähäinen parametrien määrä saattaa aiheuttaa tuoterungossa heikkoa muunneltavuutta.

3.5 Työkalupohjaiset menetelmät

Työkalupohjaisissa menetelmissä kysymyksessä on yleensä sovelluksen kääntämistä edeltävästä muunneltavuuden realisoimisesta. Käytännössä työkalulla tarkoitetaan tässä yhteydessä sovelluksen konfiguroinninhallintaan käytettäviä työkaluja [9, luku 5.2]. Työkalujen käyttö, kuten esimerkiksi esikäsittelijöiden käyttö on yksi käytetyimpiä menetelmiä tuoterunkoarkkitehtuuriin liittyvän variaation toteutuksessa [9, 62].

Luontityökalun (build-system) tehtävänä on hoitaa sovelluksen luontiin liittyviä toimenpiteitä, kuten esimerkiksi generointeja, lähdekoodin kääntöä, asennuspakettien luontia sekä tarvittavien testien suorituksia [9, luku 5.2]. Yksinkertaisimmillaan luontityökalu voi olla skripti, joka suorittaa edellä mainitut toimenpiteet ennalta määrätyssä järjestyksessä lähdekoodin käännön yhteydessä. Edistyneemmillä työkaluilla on tarjolla enemmän mahdollisuuksia vaikuttaa esimerkiksi mukaan integroitavien kirjastojen versioihin, erilaisten riippuvuuksien määrittelyihin sekä päivityksiin. [9, luku 5.2] Lisäksi

edistyneemmät työkalut kykenevät antamaan yksityiskohtaisen raportin tehdyistä toimenpiteistä.

Linux käyttöjärjestelmän koontiin käytettävä KBuild on esimerkki, jossa koostamisessa käytetään hyväksi luontityökalua [9, luku 5.2]. Sen toiminta perustuu luontiskripteihin, jotka määrittelevät, mitä kooditiedostoja kääntämiseen otetaan mukaan. Luontiskriptit suoritetaan valinnan jälkeen määritettyjen ehtojen mukaisesti.

Luontityökalulla saadaan aikaiseksi käännösvaiheen jälkeinen muunneltavuus. [9, luku 5.2] Tyypillisesti luontityökalujen variaatio perustuu määriteltäviin parametreihin, joiden arvoja tarkastellaan käännösvaiheessa. Luontityökalun hyödyntäminen on riippumaton käytetystä ohjelmointikielestä ja se soveltuu tarvittaessa myös pelkkien tiedostojen varioimiseen.

Luontityökaluun pohjautuva muunneltavuusstrategia on hyvä valinta silloin, mikäli ominaisuudet voidaan peilata esimerkiksi suoraan puhtaisiin tiedostoihin. [9, luku 5.2] Ongelmia voi kuitenkin syntyä, mikäli jotkin ominaisuudet läpileikkaavat useassa tiedostossa. Tällöin muutoksen ilmentyessä joudutaan ylikirjoittamaan kaikki läpileikkauksen sisältävät tiedostot uudelleen.

Esikäsittelijä on työkalu, joka kykenee manipuloimaan koodia ennen sen kääntöä [9, luku 5.3]. Esikäsittelijälle määritellä ehtoja, joiden perusteella se valitsee, mitä koodia käännösvaiheeseen sisällytetään mukaan. Ehdot määritetään tyypillisesti *#ifdef* ja *#endif* mukaisilla määreillä. Esikäsittelijä tarkastaa tällöin määritellyt ehdot ja jättää kääntämisen yhteydessä pois deaktivoidut ominaisuudet.

Esikäsittelijät ovat tehokas työkalu käännösvaiheessa realisoitavan muunneltavuuden toteuttamiseen ja niiden toimintalogiikka on helppo ymmärtää [62]. Ongelma esikäsittelijöiden käytössä on, samoin kuin ehtolauseisiin perustuvassa parametrisoinnissa, että ne aiheuttavat pahimmillaan vaikeasti ymmärrettävää ja ylläpidettävää koodia [9, 62]. Lisäksi esikäsittelijöiden käyttö rikkoo *Separation of concerns* –suunnittelukäytäntöä, sillä ehdolliset määreet voivat läpileikata useampaa osaa lähdekoodista, jolloin ominaisuuksia ei saada jäljitettyä omiin yksilöityihin moduuleihinsa [62].

3.6 Aspektipohjainen sovelluskehitys

Aspect Oriented Programming, eli aspektipohjainen ohjelmointi on alkujaan Yhdysvalloissa toimivan Xerox Palo Alto tutkimuskeskuksessa kehitetty paradigma, jossa tarkoituksena on modularisoida lähdekoodissa läpileikkaantuvia ominaisuuksia [9, 63, 64]. Esimerkkejä tällaisista ominaisuuksista ovat lokiin kirjoitus, käyttäjien autentikointi sekä virhetarkastelut [64]. Aspektipohjaisessa sovelluskehityksessä läpileikkaavat ominaisuudet lisätään osaksi sovelluksen lähdekoodia sen kirjoittamisen jälkeen joko staattisesti ennen kääntöä tai sitten dynaamisesti ohjelman käynnistyksen yhteydessä [9, 63].

Näin ollen usein toistuvaa sekä monessa paikkaa läpileikkaavaa koodia ei tarvitse kirjoittaa manuaalisesti käsin. Monilla ohjelmointikielillä tuki aspektipohjaiselle ohjelmoinnille on toteutettu kielilaajennusosien avulla. Näistä esimerkkejä ovat Javalle löytyvä AspectJ sekä .Net komponenttikirjaston kanssa yhteensopiva Postsharp [65, 66].

Aspektipohjaiseen ohjelmointiin liittyvät seuraavat termit [9, 67, 68]:

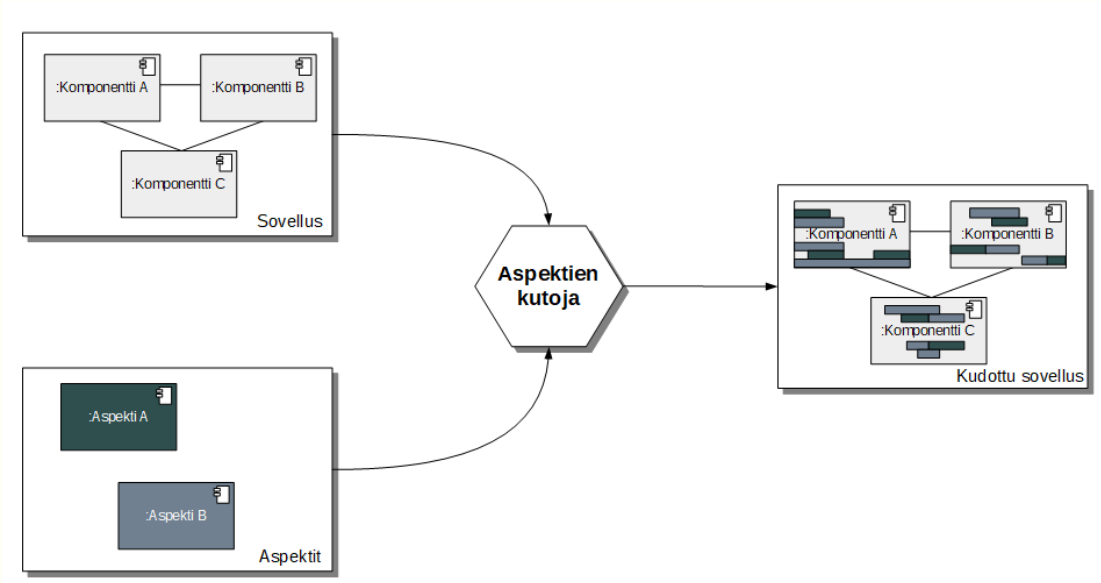
Liitospiste (*join point*) on ohjelman suorituksen aikana esiintyvä tapahtuma, jonka yhteyteen aspekti voidaan mahdollisesti kutoa. Liitospiste voi olla esimerkiksi jokin koodissa esiintyvä metodikutsu tai poikkeuskäsittely.

Neuvo (*advice*) määrittelee liitospisteessä suoritettavan toimenpiteen sekä sen, missä vaiheessa liitospistettä toiminto suoritetaan. Suoritusvaihe voi esimerkiksi esiintyä ennen liitospistettä (*before*), liitospisteen jälkeen (*after*) tai liitospisteen ympärillä (*around*).

Leikkauspiste (*pointcut*) on predikaatti, joka määrittelee, mihin liitospisteeseen läpileikkaava toiminnallisuus yhdistetään. Leikkauspisteen avulla voidaan esimerkiksi päättää, että tietyn nimisten metodien yhteydessä suoritetaan tietty toiminnallisuus.

Aspekti (*aspect*) käsittää kokonaisuudessaan leikkauspisteet sekä neuvot, muodostaen läpileikkaavan ominaisuuden toteutuksen. Aspektin avulla läpileikkaava ominaisuus on mahdollista modularisoida. Aspektien kutojan avulla aspekti saadaan yhdistettyä osaksi sovellusta.

Aspektien kutojan (*aspect Weaver*) tehtävänä on yhdistellä toistuvat aspektit kuvan 18 tavoin, osaksi kantasovellusta. Liitospisteiden avulla kutoja tietää, mihin kohtiin aspekti tulee lisätä.



Kuva 18. *Aspektien kutominen. Mukailtu lähteestä [9, luku 6.2]*

Tuoterungoissa aspektien avulla voidaan eristää myös läpileikkaantuvat ominaisuudet vaihtoehtoisesti valittaviksi piirteiksi. Tällöin loogisesti yhtä aspektia vastaa yksi ominaisuus. [9, luku 6.2] Aspektipohjaisen sovelluskehityksen avulla sovelluksen ominaisuuksia saadaan tarvittaessa laajennettua siitäkin huolimatta, vaikka sen arkkitehtuurissa ei olisi varauduttu ominaisuuksien kasvamiseen.

Aspektipohjaista ohjelmointia on kritisoitu siitä, että mikäli sopivia kehitystyökaluja ei ole saatavilla, aspekteja sisältävän sovelluksen suoritusjärjestyksen ymmärtäminen voi pelkästään lähdekoodia lukemalla olla hankalaa [69]. Lisäksi aspektipohjaista ohjelmointia on kritisoitu niin kutsutun särkyvän leikkauspisteen (*fragile pointcut*) ongelmasta [70]. Kyseinen ilmiö realisoituu esimerkiksi siinä vaiheessa, kun leikkauspisteet on määritetty koskemaan liitospisteitä kuten metodeja, jotka käyttävät tietynlaista nimikäytäntöä. Tällöin mikäli metodien nimityksissä tapahtuu muutoksia, eivät leikkauspisteet ja sitä kautta toimintojen lisäykset enää toimi. Samalla tavoin ongelmia voi syntyä myös silloin, mikäli vielä kutomattoman sovelluksen metodeissa käytetään vahingossa liitospisteille varattuja nimityksiä. Tällöin leikkauspiste sitoo neuvon sellaisen metodin yhteyteen, johon sitä ei alun perin ollut tarkoitus laittaa.

4. TUOTERUNKOARKKITEHTUURIN SUUNNITTELU

Tässä luvussa käydään läpi kohteena olevan sovellusperheen tuoterunkoarkkitehtuuria koskevan suunnittelun keskeisimmät ratkaisut. Keskiössä ovat olemassa olevan tuoteperheen variaation analysointi ja tämän pohjalta variaatiota edesauttavien suunnitteluratkaisujen tunnistaminen. Lisäksi mahdollinen tuoterunkoon siirtyminen otetaan tarkasteluun.

4.1 Kohdesovellus

Näyttämömekaniikan toiminnanohjaussovellus on PC:llä ohjattava hallintasovellus, jonka avulla teatteriympäristössä työskentelevän operaattorin on mahdollista määritellä lavasteita liikuttavien mekatronisten laitteiden toimintasekvenssejä. Toimintasekvenssejä on mahdollista ohjelmoida sekä suorittaa näytelmäkohtaisesti, jolloin esimerkiksi lavasteiden vaihto voidaan suorittaa tehokkaasti kohtauksesta toiseen siirryttäessä. Operaattorilla on mahdollista vaikuttaa, mitä laitteita yksittäiseen toimintoon sisällytetään, sekä millä fyysisellä ohjaimella ohjelmoitu toiminto aktivoidaan.

Ohjaussovellus toimii koko verkottunutta teatterijärjestelmäkokonaisuutta ohjaavana elimenä ja sen voidaan kuvata olevan luonteeltaan hyvin tehtäväkriittinen sovellus. Toimintojen on tapahduttava oikea-aikaisesti, jotta lavasteiden siirtäminen tapahtuisi näytelmän aikataulun edellyttämällä tavalla.

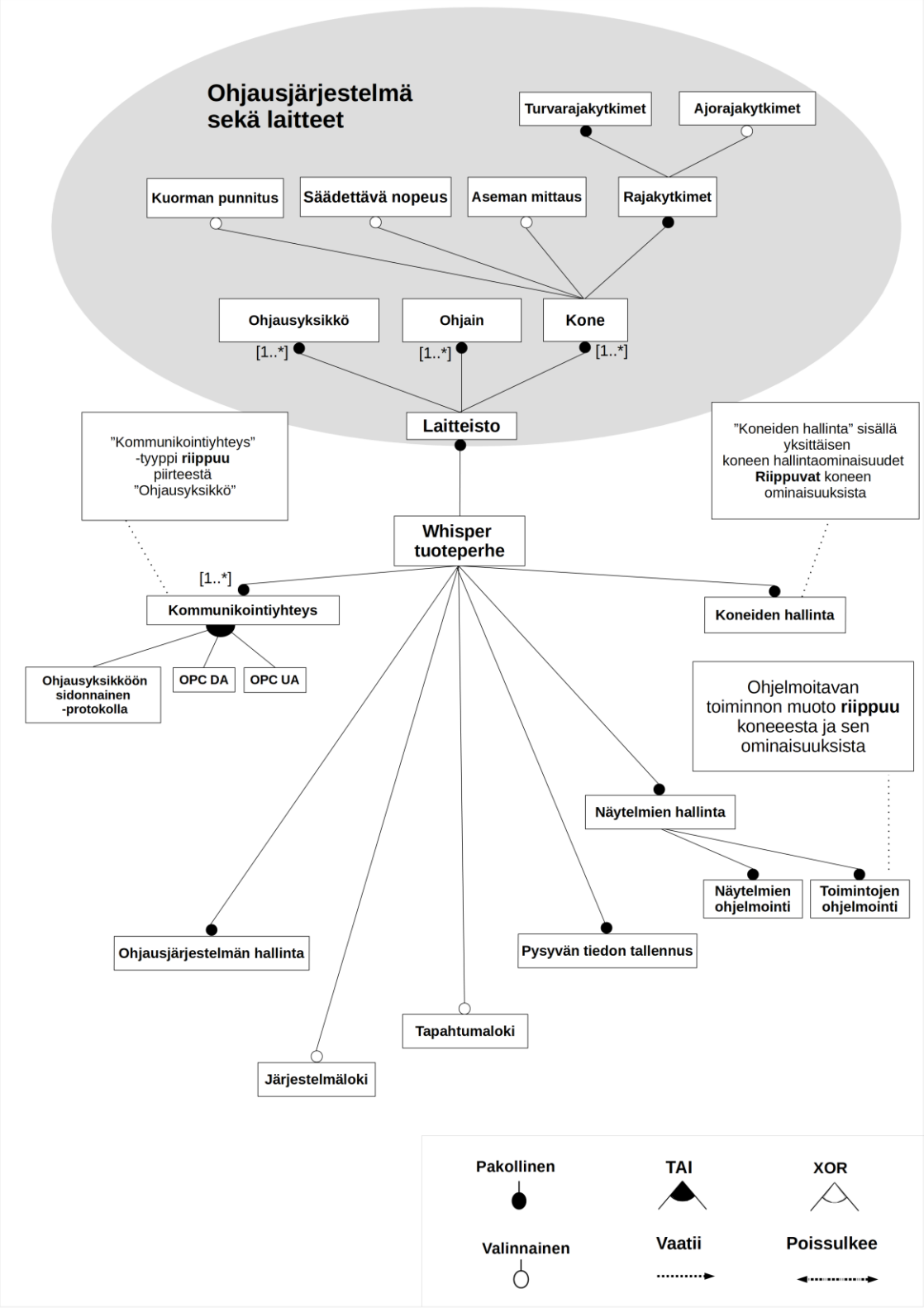
4.2 Analyysi

Valittujen muunneltavuusmekanismien tulee tukea mahdollisimman hyvin tuoterungolle asetettuja muunneltavuusvaatimuksia, jonka vuoksi tuoteperheen variaatio tulisi tunnistaa oikein. Variaatiopisteiden tunnistamista varten, kohdesovellusperheen tämän hetkisestä muodosta luotiin malli. Samankaltaisten sekä muuttuvien piirteiden analysoinnissa tutkittiin kuutta jo toimitettua teatterijärjestelmäkokonaisuutta sekä niissä käytössä olevaa näyttämömekaniikan ohjaussovellusta. Sovelluksien käyttötapauksia sekä ominaisuuksia vertailemalla merkittävimmät piirteet taulukoitiin ja niiden pohjalta muodostettiin kuvan 19 mukainen piirremalli, joka käsittää tuoteperhettä koskevan merkittävimmän ydintoiminnallisuuden sekä mahdolliset vaihtoehtoiset piirteet. Kyseessä on kardinaliteettinen piirremalli, joka esittää tarvittaessa myös toistuvien piirteiden esiintymislukumäärät [71]. Mallinnuksen tavoitteena on tunnistaa ne oleellisimmat muuttuvat piirteet, joihin huomioon kiinnittämällä sovellusperheen arkkitehtuurista saataisiin mahdol-

lisesti modulaarisempi, jolloin projektikohtainen muunneltavuus onnistuisi tulevaisuudessa tehokkaammin.

4.2.1 Variaation tunnistus

Analyysi pohjautuu tuoteperheestä mallinnettuun piirremalliin (kuva 19). Ensisilmäyksellä piirremallin pohjalta on havaittavissa, että merkittävin variaatio ilmenee ohjausjärjestelmän sekä laitteistopuolen piirteissä: järjestelmässä olevien koneiden määrä ja niiden sisältämät ominaisuudet, koneiden operointiin käytettyjen ohjainten määrä sekä liikkeenohjauksesta vastuussa olevien ohjausyksiköiden määrä ja tyyppi voivat muuttua toimitetusta sovelluksesta riippuen. Lisäksi lokitoiminnallisuus voi esiintyä sovelluksesta riippuen ehdollisena.



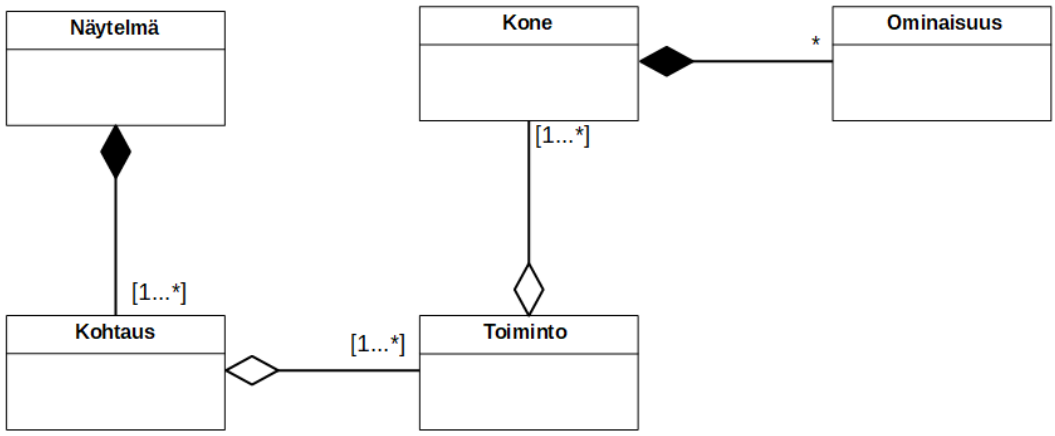
Kuva 19. *Whisper-tuoteperhettä kuvaava piirremalli*

Piirremallin pohjalta on siis tunnistettavissa tärkeimmät muuttuvat kohteet, joiden pohjalta voidaan lähteä miettimään sopivia keinoja muunneltavuuden toteutusta varten. Mallin pohjalta tunnistettuja merkittävimpiä muuttuvia piirteitä ovat siis seuraavat:

Kommunikointiprotokollan variaatio. Sovelluksessa käytetty kommunikointiprotokolla on suoraan riippuvainen ohjausjärjestelmässä käytetystä ohjausyksiköstä, joita voi määrällisesti esiintyä tilanteesta riippuen enemmän kuin yksi. Muuttuvan ohjausyksikön tyyppi vaikuttaa suoraan sovelluksessa käytettävään kommunikointiprotokollaan. Kysymyksessä on sisäinen variaatio, sillä käytetty kommunikointiprotokolla ei suoranaisesti näy suoraan sovelluksen loppukäyttäjälle. Lisäksi kysymyksessä on variaatiosta, joka tulisi ottaa huomioon tuoterungon ajan suhteen tapahtuvassa muutoksessa: mikäli suunnitellun tuoterungon elinkaaren nähdään ylettyvän kymmenien vuosien päähän tulevaisuuteen, tulee kommunikointiprotokollan olla joustavasti muunneltavissa, ilman koko sovelluksen kattavaa uudelleen räätälöintiä.

Ohjausjärjestelmän laitteiden määrä voi vaihdella. Ohjausjärjestelmässä esiintyvät laitteet voivat eri sovelluksissa esiintyä eri määreissä sekä erilaisissa muodoissa. Järjestelmät sisältävät kukin esimerkiksi vaihtelevan määrän ohjaimia, ohjausyksiköitä sekä koneita. Vaihtuvat laitemäärät varioivat sekä sovelluslogiikkaa, että käyttöliittymän ulkoasua. Ohjausjärjestelmän sovellukseen peilaava variaation vaikutus on luonteeltaan vahvasti ulkoista.

Yksittäinen lavasteita liikuttava kone voi sisältää erilaisia ominaisuuksia (kuorman punnitus, nopeuden asetus, aseman mittaus, erilaiset rajakytkimet). Vaihtuvilla ominaisuuksilla on merkittävä vaikutus sovelluksen rakenteeseen. Variaation vaikutus näkyy suoraan esimerkiksi koneiden hallintaan liittyvissä ominaisuuksissa sekä sovelluksen tärkeimmän toimintalogiikan, eli näytelmäkohtaisten toimintasekvenssien ominaisuuksissa. Riippuvuusketju etenee kuvan 20 mukaan kerroksittain. Yksittäisen toiminnon sisälle voidaan sisällyttää koneita, jotka koostuvat erilaisista ominaisuuksista. Koneilla suoritettavat toiminnot voidaan sittemmin koostaa osaksi kohtauksia, joiden pohjalta rakennetaan sovelluksella ajettavat näytelmäsekvenssit.



Kuva 20. *Näytelmien sisältämien kohtausten relaatio toimintojen kautta koneiden ominaisuuksiin.*

Konetta koskevat vaihtuvat ominaisuudet ovat siis yksi merkittävimmistä variaatiopisteistä, sillä sen muoto vaikuttaa hyvin vahvasti sovelluksen tärkeimpään toimintalogiikkaan. Muunneltavuus on tyypiltään ulkoista variaatiota, sillä se vaikuttaa suoraan sovelluksen käyttötapauksiin.

Lokitoiminnallisuus. Sovellukseen voi vaihtoehtoisesti kuulua yksityiskohtaista lokia keräämiä toiminnallisuuksia, kuten tapahtumaloki sekä järjestelmäloki. Ominaisuudet on piirremallissa määritelty valinnaisiksi, joka tarkoittaa, että niiden tulisi tarvittaessa olla mahdollista ottaa käytännössä käyttöön valinnaisesti. Lokitoiminnallisuus on luonteeltaan sovellusta useassa paikassa läpileikkaavaa.

Sovelluksen toimintalogiikka sekä käyttötapaukset perustuvat pitkälti koneiden sekä ohjausjärjestelmän muotoon. Lisäksi sovelluksen toimivuus on riippuvainen sovelluksen ja ohjausyksikön välisestä kommunikoinnista. Kommunikointi-ominaisuuden muoto taas riippuu suoraan ohjausjärjestelmässä käytössä olevasta ohjausyksiköstä. Näin ollen analyysin pohjalta voidaan todeta, että kohdesovelluksessa esiintyvä variaatio peilaa pitkälti ohjausjärjestelmässä sekä mekaniikassa esiintyvää variaatiota.

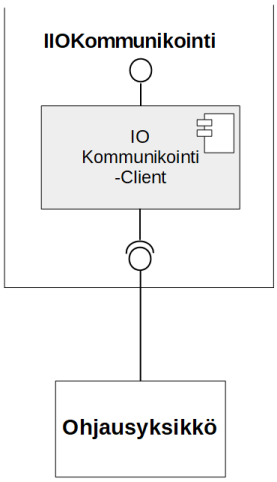
4.2.2 Soveltuvien erikoistamismenetelmien arviointi

Piirremallinnuksen kautta sovelluksen muuttuvat ja staattiset ominaisuudet on tunnistettu, jonka jälkeen variaation toteutukseen voidaan miettiä sopivimpia ratkaisuja. Luvussa 3 esitettiin useita eri keinoja tuoterunkoarkkitehtuurin muunneltavuuden realisoimiseen. Suunnitteluvaiheessa huomioon on otettava tuoterungon muunneltavuuden yli- tai alimitoitusta koskeva riski. Lisäksi toinen suunnittelua koskeva riski on se, että variaatiopisteet suunnitellaan liian monimutkaisiksi, joka hankaloittaa tuoterunkoarkkitehtuurin kokonaisvaltaista ymmärtämistä.

Suunnitteluvaiheen tarkasteluun otetaan analyysivaiheessa tunnistetut tärkeimmät muuttuvat piirteet, joita olivat kommunikointiprotokollaa, ohjausjärjestelmää, koneita sekä lokitoiminnallisuutta koskeva variaatio. Tässä aliluvussa esitetään mahdollisia yksityiskohtaisia suunnitteluratkaisuehdotuksia näiden variaatiopisteiden toteutuksille:

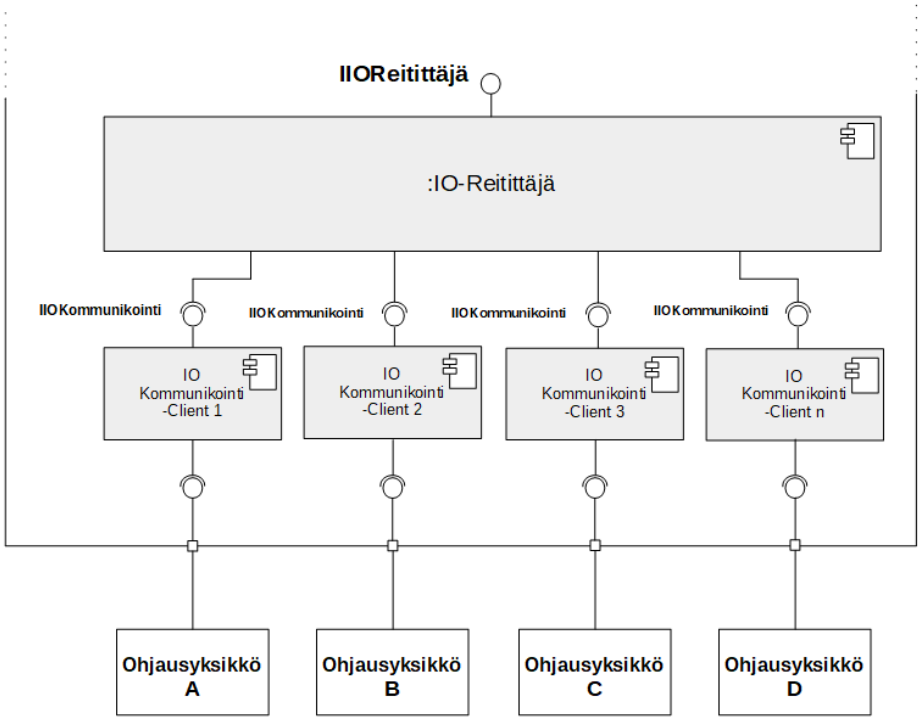
Kommunikointiprotokollaa koskeva variaatio on riippuvainen ohjausjärjestelmässä käytetystä ohjausyksiköstä. Ohjausyksikön tyyppi ja sen tukemat ominaisuudet määrittelevät pitkälti sen, mitä protokollaa kommunikoinnissa voidaan käyttää. Ohjausyksikön iällä on esimerkiksi merkittävä vaikutus mahdollisen kommunikointiprotokollan tyyppiin. Sovellus voi lisäksi kommunikoida useamman kuin yhden ohjausyksikön kanssa samanaikaisesti. Kommunikointirajapinnan kautta kohdesovelluksessa määritetyt käskyt kirjoitetaan ohjausyksikön suoritettavaksi. Nykyisessä sovellusrungossa kommunikointiprotokolla on hyvin vahvasti sidoksissa toteutukseen, jonka vuoksi sen muuttaminen tarkoittaa hyvin suurta työmäärää.

Kommunikointia koskeva varioitavuus voitaisiin vaihtoehtoisesti toteuttaa niin, että tiettyä protokollaa edustava kommunikointitoiminnallisuus sisällytettäisiin siihen tarkoitukseen tarkoitettuun ohjelmistokomponenttiin, joka tarjoaisi sittemmin sovelluksen sisäisesti standardoidun kommunikointirajapinnan (kuva 21). Näin ollen kommunikointikomponentin pystyisi vaihtamaan tarvittaessa toiseen ilman koko sovelluksen kattavaa lähdekoodin muokkausta. Mikäli sovelluksessa tarvittaisiin useampaa kommunikointiyhteyttä saman protokollan alaisuudessa, olisi kommunikointikomponentista hyvä olla mahdollista luoda ajoon useampi yksilöity instanssi.



Kuva 21. *Kommunikointikomponentti, joka kommunikoi ohjausyksikön kanssa soveltuvalla protokollalla, tarjoten sovelluksen sisällä standardirajapinnan.*

Useamman kommunikointiyhteyden vuoksi, tiedonsiirto voitaisiin abstrahoida niin, että ylemmän tason komponenttien sekä ohjausyksikön kanssa kommunikoivien komponenttien välinen kommunikointi tapahtuisi yhteisen reitittäjäkomponentin kautta. Reitittäjäkomponentin tehtävänä olisi välittää ylemmän tason komponenttien kommunikointipyynnöt kohteen perusteella halutun ohjausyksikön kanssa kommunikointia suorittavan palvelukomponentin instanssille. Valittu palvelukomponentti kommunikoisi sitten ohjausyksikön kanssa soveltuvaa protokollaa käyttäen (kuva 22). Tällä tavoin kommunikointi-rajapinta saataisiin abstrahoitua, jolloin ylemmän tason komponenttien ei tarvitsisi olla riippuvaisia yksittäisten kommunikointikomponenttien toteutuksista.



Kuva 22. *Reitityskomponentti kommunikointiyhteyden abstrahoijana.*

Ohjausjärjestelmän laitteiden määrä voi vaihdella (Koneet, Ohjaimet). Ohjasjärjestelmässä vaihtuva laitemäärä edellyttää, että tuoterungon sovellusrunko mukautuisi tilanteeseen joustavasti. Ominaisuuden joustava käyttöönottaminen edellyttää, että sovellukseen tulisi olla mahdollista määritellä, kuinka paljon mitäkin laitteita sovellukseen sisällytetään.

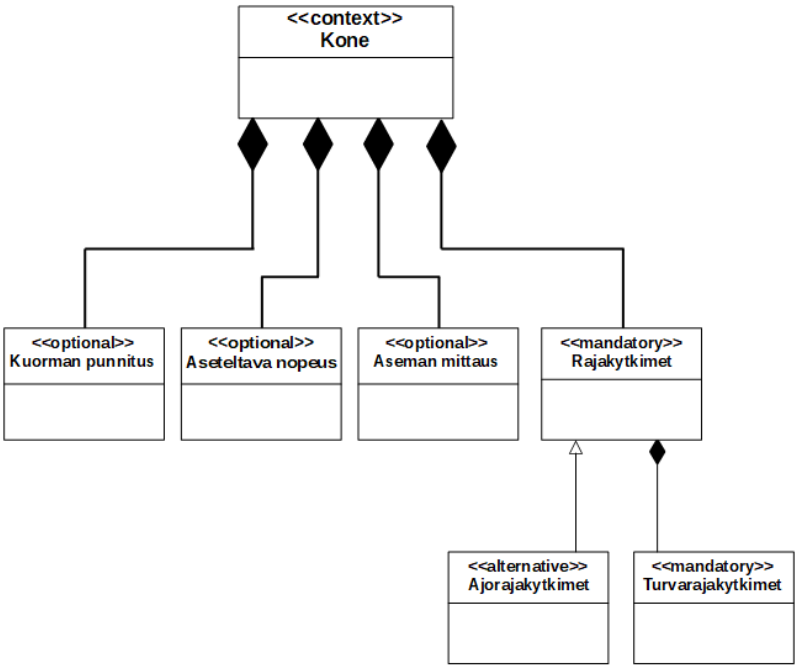
Menetelmän toteuttamisessa mahdollisuuksia voisivat olla esimerkiksi dynaaminen instanssien alustaminen tai vaihtoehtoisesti staattinen koodin generointi. Molemmissa tapauksissa laitemäärää koskevaan variaatioon vaikutettaisiin parametrien avulla.

Staattisessa ratkaisussa laitemäärää koskeva koodi generoitaisiin siihen tarkoitusta varten räätälöidyllä työkalulla. Työkalun avulla voisi esimerkiksi valita, minkä tyyppisistä laitteista on kysymys ja montako laitetta generointiin sisällytetään. Variaatio olisi tällaisessa toteutuksessa tyypiltään käännöstä edeltävää, jolloin muutoksien tekeminen tarkoittaisi uudelleen generointia.

Dynaamisessa versiossa koneiden määritys voitaisiin tehdä samalla tavoin ulkoisen työkalun avulla. Sen sijaan, että parametrien pohjalta generoitaisiin käännettävää koodia, varioitaisiin niiden pohjalta suorituksen aikaisten instanssien muotoa. Variaatio sidottaisiin tällaisessa tapauksessa käynnistyksen tai muun sopivan alustustilanteen yhteydessä.

Yksittäinen kone voi sisältää erilaisia ominaisuuksia (kuorman punnitus, säädettävä nopeus, aseman mittaus, rajakytkimet). Ohjausjärjestelmässä esiintyvien koneiden vaihtuvat ominaisuudet ovat sovelluksen rakenteeseen merkittävällä tavalla vaikuttava

piirre. Yksittäisessä, lavasteita liikuttavassa koneessa voi olla kytkettynä esimerkiksi kuormaa punnitseva anturi sekä erilaisia rajakytkimiä. Koneessa voi tyypistä riippuen olla mahdollisuus säätää nopeutta, jonka lisäksi se voi sisältää myös paikoitusominaisuuden, jolloin sitä voidaan ajaa haluttuihin maalipisteisiin. Käytännössä koneen fyysinen instrumentointi määrittelee, mitä ominaisuuksia siihen ohjaussovelluksessa sisällytetään. Kuvassa 23 on UML-stereotyypin piirremallin avulla havainnollistettu, millä tavoin teatterijärjestelmässä esiintyvä kone koostuu erilaisista instrumenteista. Jotta sovelluksen variaation valjastaminen onnistuisi tuoterungossa tehokkaalla tavalla, tulisi ohjaussovelluksessa esiintyvälle koneelle olla mahdollista määrittää instrumentointiin sidonnaiset ominaisuudet selkeällä tavalla.



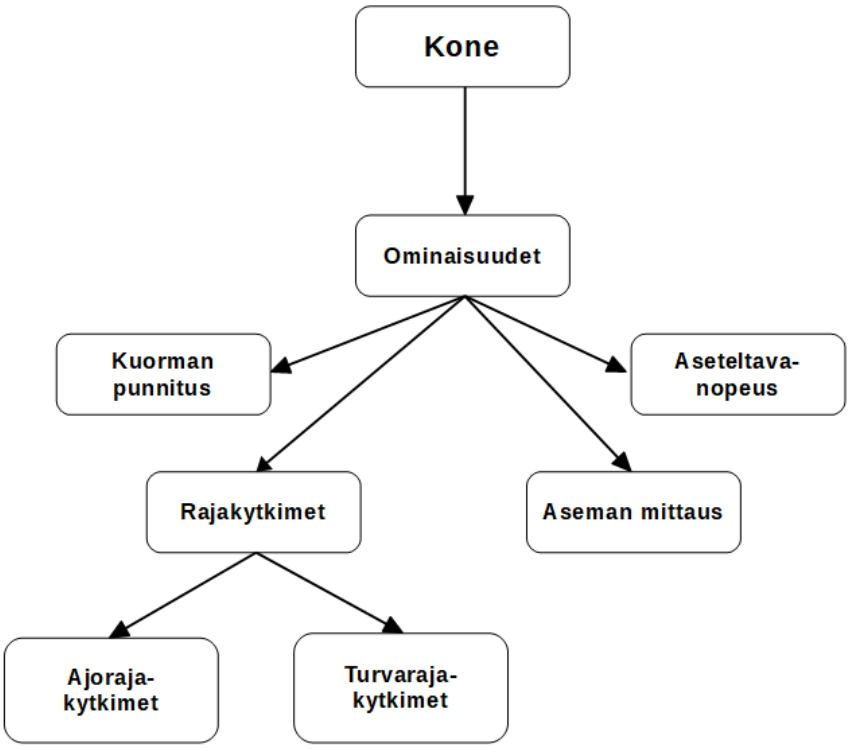
Kuva 23. *Koneeseen sisältyvät ominaisuudet piirremallin UML-stereotypialla esitettynä.*

Yleistäen koneeseen liittyvä ominaisuus tarkoittaa ohjaussovelluksen kontekstissa ohjausyksikön muistialueelle viittaavaa hallinta- tai monitorointirajapintaa, jolloin ominaisuuden varsinainen toimintalogiikka sijaitsee liikkeenohjauksesta vastuussa olevassa ohjausyksikössä. Tällöin tiettyyn ominaisuuteen on sidottuna tietyt ohjausyksikön muistialueelle kohdistuvat muuttujaviittaukset.

Käytännön toteutusta ajatellen ominaisuuksia toteuttavat piirteet voitaisiin kuvan 23. piirremallin tavoin koostaa osaksi perustoiminnallisuuden sisältävää koneobjektia. Tällöin kone sekä siihen määritetyt ominaisuudet muodostaisivat yhdessä fyysistä laitetta kuvaavan oliomallin. Kone sisältäisi näin ollen joukon kyseiseen koneeseen kuuluvia ominaisuuksia. Ominaisuudet voisivat sittemmin sisältää esimerkiksi tarvittavat ohjausyksikön muistialueelle kohdistuvat muuttujaviittaukset. Koostamalla tieto selkeästi yk-

silöitäviin kokonaisuuksiin, saataisiin sovellukseen parempaa läpinäkyvyyttä yksittäisen koneen ominaisuuksille.

Rekursiokooste – suunnittelumallin tavoin, yksittäisen koneen alaisuuteen voitaisiin luoda eri ominaisuuksista muodostuva hierarkinen rakenne. Hierarkia voisi esimerkiksi olla kuvan 24 kaltainen, jossa ominaisuudet koostettaisiin koneen kantaolion alaisuuteen. Tarvittaessa, mikäli koneeseen tulisi jotain perustoiminnallisuudesta poikkeavia ominaisuuksia voisivat ne muodostaa hierarkisassa omat haaransa. Mallia hyödyntäen, usein projekteissa esiintyville konetyypeille voisi luoda uudelleenkäytettäviä objektimalleja, joissa tietty ominaisuushierarkia löytyisi jo valmiina.

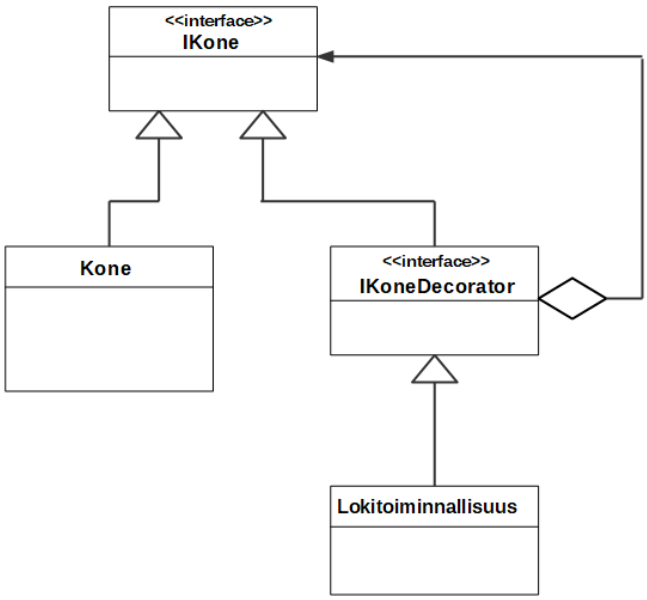


Kuva 24. *Konekohtaisten ominaisuuksien yhteen koostaminen hierarkisesti*

Lokitoiminnallisuus. Kuten analyysivaiheessa todettiin, on lokitoiminnallisuus mahdollinen vaihtoehtoinen piirre. Kyseessä on kuitenkin myös sovelluksen arkkitehtuuria useassa kohtaa läpileikkaava piirre. Tästä syystä ominaisuuden aktivointi ja deaktivointi ilman modulaarista ratkaisua tarkoittaa sitä, että kehittäjän tulee silmämääräisesti tarkastella, missä kohdissa lokitoiminnallisuutta suorittavaa funktiota/metodia on kutsuttu.

Eräs vaihtoehto ominaisuuden modularisoinnille olisi luvussa 3 esitetty aspektipohjainen sovelluskehitys. Lokitoiminnallisuus on tyypillinen esimerkki läpileikkaantuvasta ominaisuudesta ja aspektien avulla se saataisiin eristettyä omaksi vastuunalaiseksi moduuliksi.

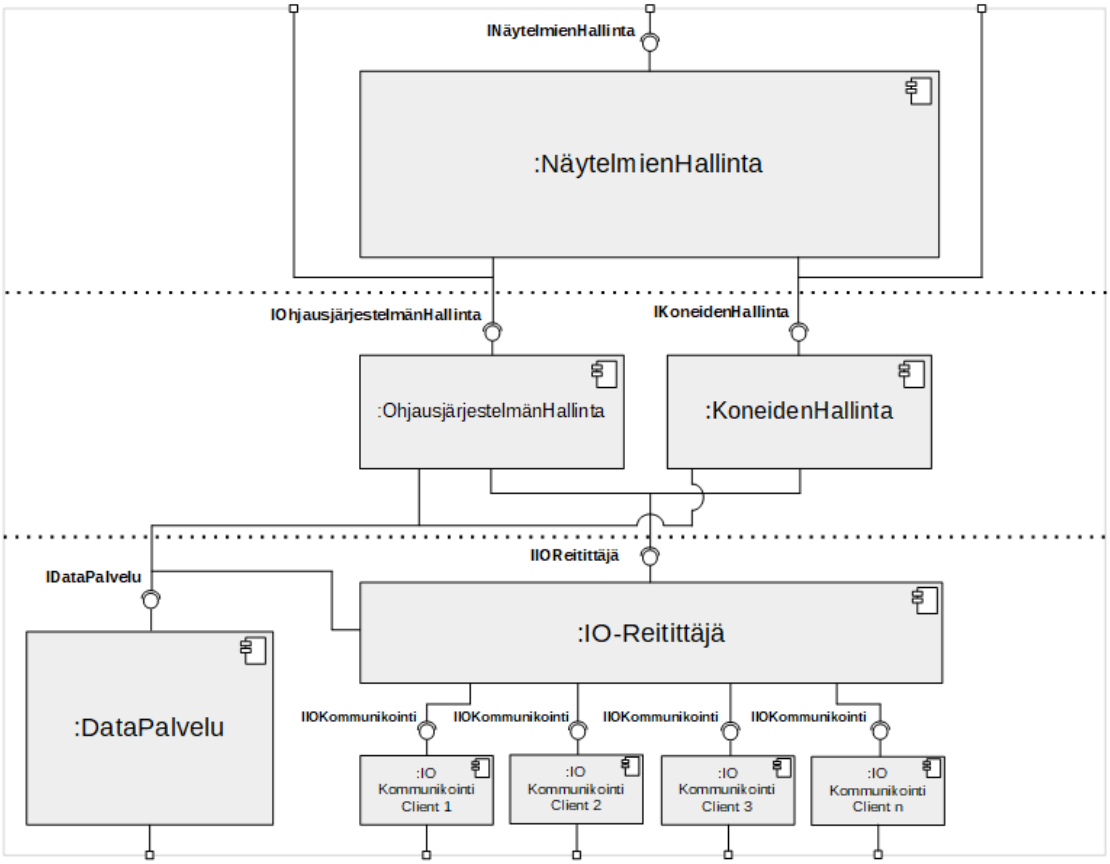
Aspektien sijaan vaihtoehtoinen menetelmä voisi olla on koristelija-suunnittelumallin hyödyntäminen (kuva 25). Suunnittelumallin avulla objektien rajapintojen yhteyteen saataisiin tarvittaessa käärittyä lokitoiminnallisuus ja samalla lokitoiminnallisuus saataisiin aspektien tavoin eristetty omaksi vastuunalaiseksi moduuliksi [72]. Lokitoiminnallisuus sidottaisiin tällaisessa toteutuksessa sovellukseen ajon aikana dynaamisesti. Ominaisuuden käyttöönotto voitaisiin tehdä siihen tarkoitetulla parametrilla.



Kuva 25. *Esimerkki lokitoiminnallisuuden lisäämisestä koneen luokkaan koristelija-suunnittelumallin avulla.*

4.2.3 Arkkitehtuurisuunnitelma

Komponenttitason arkkitehtuurisuunnitelma on esitetty kuvassa 26. Sen tarkoituksena on koostaa edellä ehdotetut yksityiskohtaiset variaatiomenetelmät osaksi korkean tason arkkitehtuuria. Esitetyssä kuvauksessa komponentit on roolitettu kuvan 19 piirremallin esittämien ominaisuuksien mukaisesti. Kysymyksessä on tiedonsiirrosta sekä sovelluslogiikasta vastaavasta kerroksesta, eli kuvauksessa ei oteta kantaa käyttöliittymäkerrokseen.



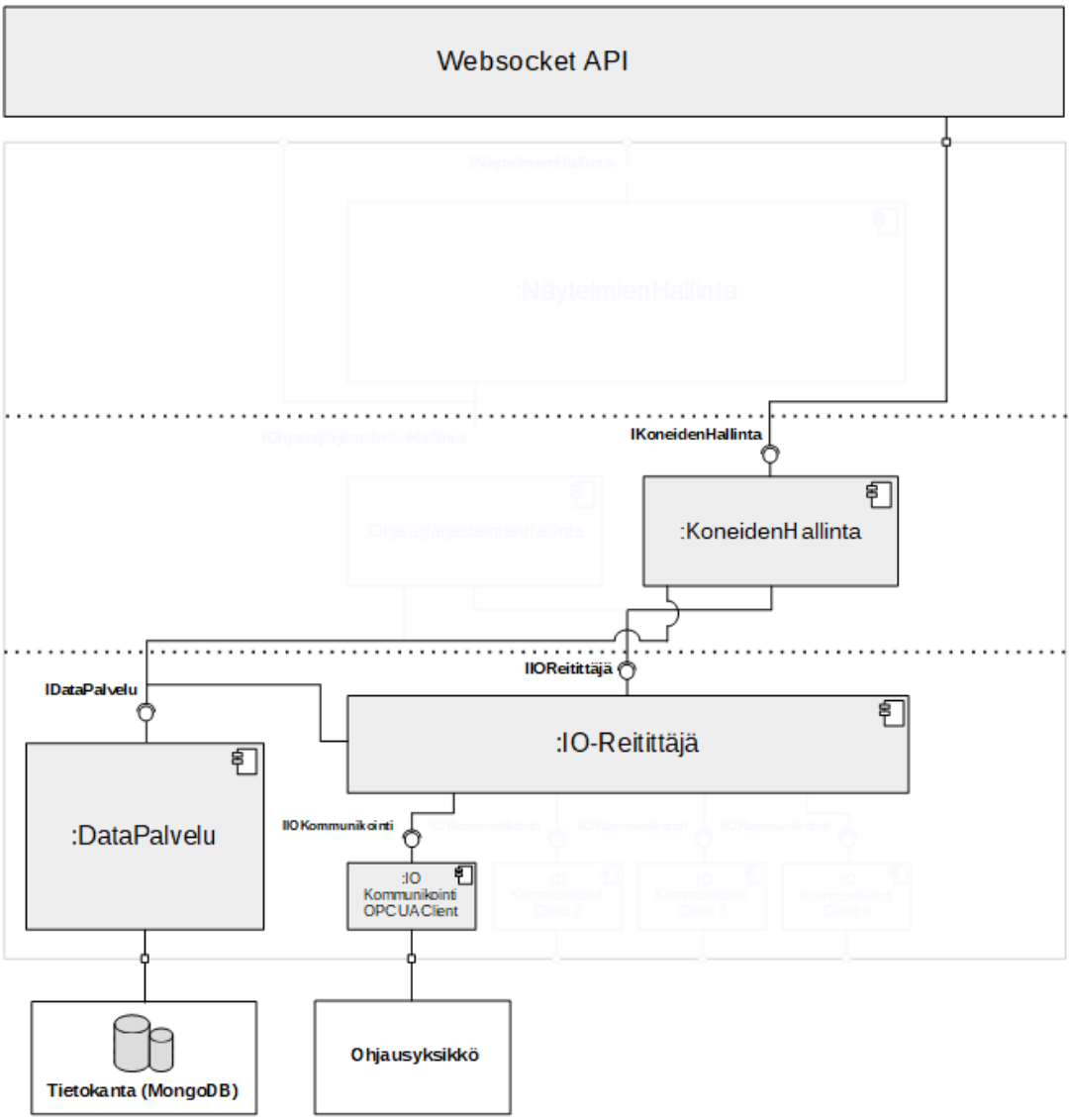
Kuva 26. *Alustava suunnitelma tunnistetun variaation huomioon ottavasta tiedonsiirto/liiketoiminta-kerroksen arkkitehtuurista.*

Esitetyssä suunnitelmassa vastuu on jaettu kolmeen eri alikerrokseen. Alimmaiselta kerrokselta löytyy tiedonsiirrosta vastaa kerros, jossa hoidetaan ohjausyksiköihin kohdistuvat kommunikointiyhteydet sekä tietokantayhteydet. Ohjausyksiköiden kommunikoinnin abstrahoijana toimii *IO-Reitittäjä* -komponentti ja pysyvän tiedon abstrahoijana *DataPalvelu*-komponentti. *OhjausjärjestelmänHallinta* sekä *KoneidenHallinta* hoitavat kukin omaan kontekstiinsa sidottujen objektien käsittelyn sekä tarjoavat rajapinnan näiden käyttöön. Päällimmäisenä esiintyy *NäytelmienHallinta* komponentti, jonka tehtävän on hoitaa näytelmäsekvenssien luontiin ja suoritukseen liittyvä liiketoimintalogiikka. Ajatuksena on, että *NäytelmienHallinta* pystyy mukautumaan ominaisuuksissaan *OhjausjärjestelmänHallinta* sekä *KoneidenHallinta* komponenttien sisällä määritettyjen objektikuvausten mukaan, sillä kuten kuvassa 20 esitettiin, määräytyvät sen ominaisuudet pitkälti koneiden tukemien ominaisuuksien mukaisesti.

Esitetyssä arkkitehtuurissa tavoitteena on eristää toiminnallisuus loogisesti, niiden suorittavien palveluiden mukaisesti. Arkkitehtuurissa modulaarisuutta edesautetaan käyttämällä luokkien keskinäisessä vuorovaikutuksessa staattisia rajapinta-määrittelyitä.

4.3 Demo-sovellus

Edellisessä aliluvussa suunniteltuja ratkaisuja testattiin käytännössä toteuttamalla rajatun toiminnallisuuden sisältävä demo-sovellus. Sovellus koostuu IO-kommunikoinnista, IO-palveluiden reitittäjästä, tietokanta-yhteyden hoitavasta datapalvelusta sekä koneiden hallintakomponentin prototyyppitoteutuksista (kuva 27). Sovellus toteutettiin Typescript-kielellä Node.js suoritusympäristöön ja se tarjoaa ulospäin Websocket-protokollan kautta rajapinnan sovelluksen käyttöön.



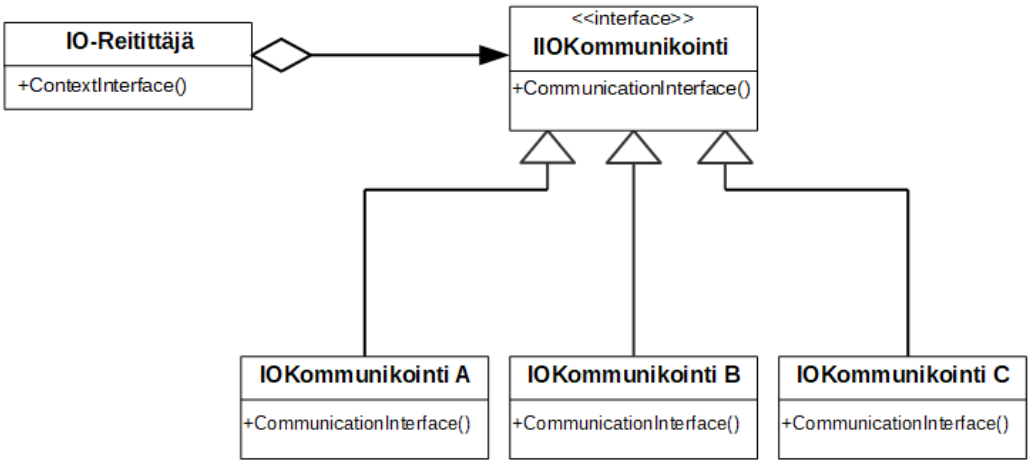
Kuva 27. *Työn puitteissa toteutetut prototyyppisovellus.*

Demo-sovelluksen päätarkoituksena on havainnollistaa edellisessä kappaleessa ehdotettujen variaatiopisteiden toimivuutta. Vaikka demo-sovelluksessa ei toteutettu kokonaisuudessaan edellisessä kappaleessa esitettyä arkkitehtuuria, ovat toteutetut komponentit analyysivaiheessa tunnistetun variaation kannalta merkittäviä. Näin ollen ratkaisuja ana-

lysoimalla voidaan tehdä johtopäätöksiä siitä, millä tavoin variaation huomioonottava suunnittelu edesauttaa tuoterunkoarkkitehtuurin muunneltavuutta. Lisäksi, vaikka testisovellus rakennettiin Typescript-kielellä Node.js suoritusympäristöön, halutaan variaatiopisteiden toteutuksien olevan muillekin oliopohjaisille ohjelmointikielille yleistettävissä. Tämä johtuu siitä, että mikäli tuoterungon todellisessa toteutuksessa päädytään jatkossa toisenlaiseen teknologiseen alustaan ja ratkaisuun, variaatiomenetelmä olisi yhä sovellettavissa.

4.3.1 Variaatiopisteiden käytännön testaus

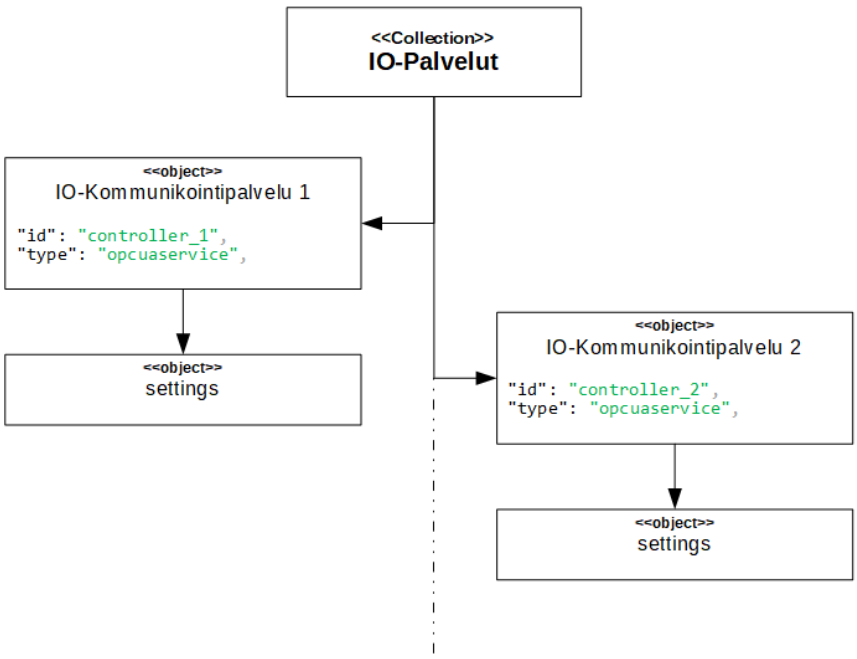
Kommunikointiprotokollan variaatio toteutettiin edellisen aliluvun suunnitelmaa mukaillen. Demo-sovelluksessa eri kommunikointiprotokollaa käyttävät komponentit jakavat yhteisen sovelluksen sisäisen standardirajapinnan *IIOKommunikointi*. Ylemmän tason viestien reitittämisestä vastaa IO-Reitittäjä -komponentti, jonka sisälle kommunikointikomponenttien instanssit alustetaan (kuva 28). Toteutus mukailee Strategia-suunnittelumallin mukaista ratkaisua.



Kuva 28. *IO-Reitittäjä -luokkaan sisällytetään joukko kommunikointistrategioita.*

Ajon aikaisten kommunikointikomponenttien instanssien alustamiseen on mahdollista vaikuttaa tietokantaan tallennettujen alustusparametrien kautta (kuva 29). Tietokanta sisältää kommunikointi-instanssien määrää ja muotoa määrittelevän joukon (*collection*). JSON-muodossa esitetyissä objektikuvauksissa kommunikointistrategioille määritetään avaimena toimiva ID-nimi, kommunikointityyppi sekä kyseisiin kommunikointiprotokollaan sidotut mahdolliset asetukset. Tyyppi-parametrin perusteella, alustusvaiheessa instanssien luomisesta vastaava tehdas-luokka osaa palauttaa oikeanlaisen kommunikointiluokan instanssin. ID toimii avaimena, jonka perusteella kommunikointipalveluita

käyttävät ylemmän tason komponentit voivat viitata haluamaansa kommunikointi-instanssiin.



Kuva 29. *IO-palvelukomponenttien instanssien määrittäminen tietokantaan.*

Yksittäiseen kommunikointikomponentin instanssiin määritetään lisäksi käytetyn protokollan edellyttämin tavoin listaus tarvittavista muuttujaviittauksista. Muuttujalistassa yksittäiselle muuttujalle määritetään ohjausyksikön muistialueelle viittaava nimi sekä sovelluksen sisällä käytettävä ristiviittausnimi. Demo-sovelluksessa toimiva kommunikointikomponentti rakennettiin OPC UA-asiakaskomponentin avulla, jolloin ohjausyksikön muistialueelle kohdistuva muuttujaviittaus tapahtuu node-id:n avulla. Kuvassa 30 on esitetty JSON-muodossa esimerkki muuttujalistauksen toteutuksesta.

Muuttujalistauksen pohjalta palvelukomponentti rekisteröityy monitoroimaan sille määritetyn OPC UA-palvelimen muuttujia. Tieto siirtyy UA-palvelulta tarkkailija-suunnittelumallia mukaillen asynkronisesti, jolloin arvoja siirretään ainoastaan siinä vaiheessa, kun niissä on tapahtunut muutoksia. Muutoksien tapahtuessa muuttunut arvo kirjoitetaan eteenpäin ristiviittausmuuttujaan, jonka avaimena toimii *ioservice_tagname.* Avaimeen viitaten, ylemmän tason komponentit voivat rekisteröityvät kuuntelemaan näitä muuttujia. Takaisinkirjoitus on mahdollista tehdä viittaamalla haluttuun kommunikointi-instanssin ID:hen ja antamalla parametrina haluttu muuttujan ristiviittausnimi sekä kirjoitettava arvo.

```
{
  "variablelist": [
    {
      "nodeid": "ns=3;s=\"DB1000\".\"PN1_Asema\"",
      "ioservice_tagname": "PN1_Asema",
      "datatype": "INT32",
      "comment": "PN1 mitattu asema"
    },
    {
      "nodeid": "ns=3;s=\"DB1000\".\"PN1_Maali\"",
      "ioservice_tagname": "PN1_Maali",
      "datatype": "INT32",
      "comment": "PN1 asetettu maali"
    },
    {
      "nodeid": "ns=3;s=\"DB1000\".\"PN1_Nopeus\"",
      "ioservice_tagname": "PN1_Nopeus",
      "datatype": "INT32",
      "comment": "PN1 nopeus asetusarvo"
    },
    ...
  ]
}
```

Kuva 30. *Muuttujien viittauslista JSON-muodossa esitettynä.*

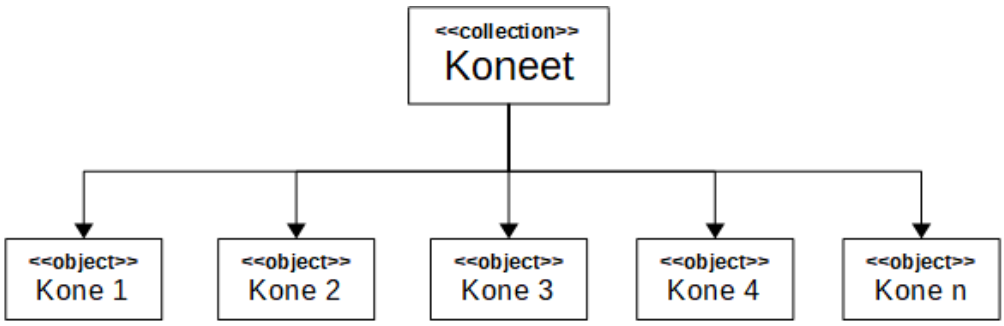
Kommunikointistrategiasta riippuen, ohjausyksikön muistialueelle tapahtuva muuttujaviittaus voi esiintyä ohjausyksiköstä riippuen erilaisena. Näin ollen, mikäli ohjausyksikön muistialueelle viittaavaa muuttujanimitystä käytettäisiin sovelluksen ylemmillä kerroksilla suoraan, jouduttaisiin kommunikointistrategiaa muuttaessa vaihtamaan myös kaikki ylemmillä kerroksilla esiintyvät muuttujaviittaukset uusiksi.

Ristiviittausnimeen perustuva muuttujaviittaustyyli mahdollistaa kuitenkin sovellukselle tehtävän kommunikointikomponentin variaation ilman edellä mainittua lisätyötä. Esimerkiksi, mikäli kommunikointistrategia vaihdettaisiin yksittäisessä sovelluksessa toiseen tavoin, että ainoastaan ohjausyksikön muistialueelle osoittavat muuttujanimet muuttuisivat, voisivat ristiviittausnimitykset pysyä edelleen samanlaisina. Näin ollen sovelluksen ylemmissä kerroksissa ei tarvitsisi tehdä kommunikointistrategian vaihtamisen yhteydessä muutoksia.

Koneiden määrää ja ominaisuuksia koskeva variaatio. Nykyisessä sovellusrungossa, koneita koskevat muutokset toteutetaan staattisesti lähdekoodiin, käsin muokaten. Aikaa muutoksiin voi kokonaisuudessaan mennä testauksineen useita päiviä.

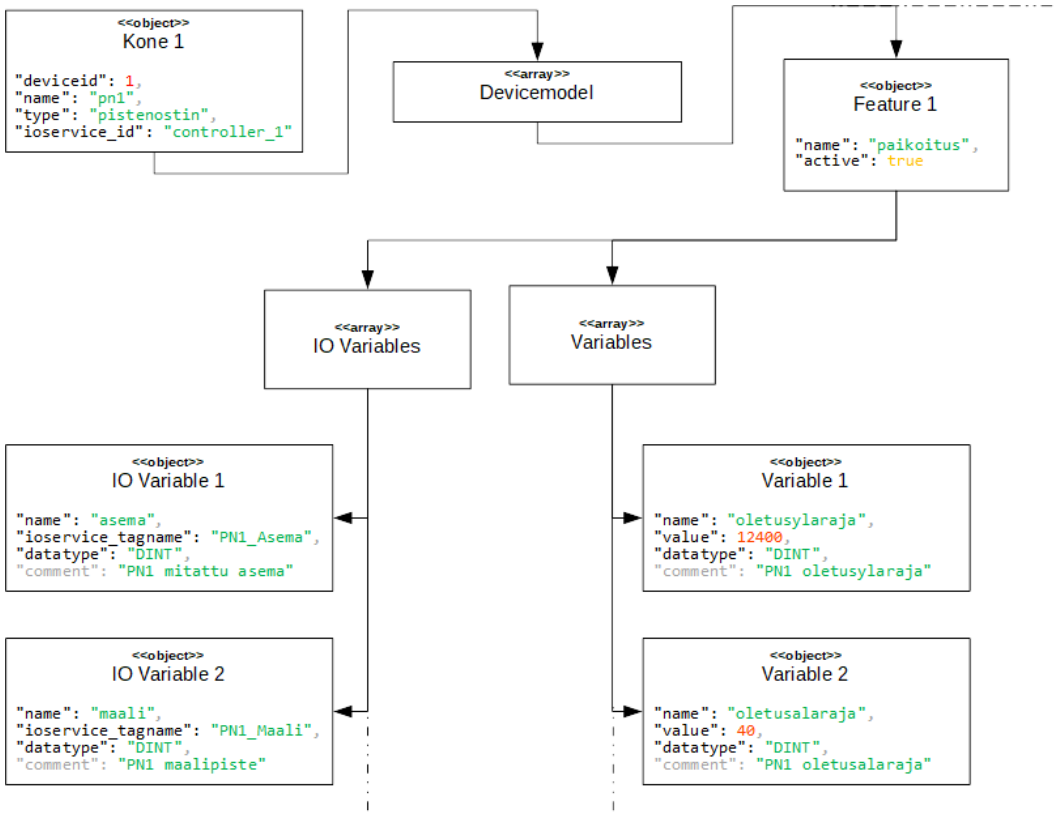
Demo-toteutuksessa koneiden lisäys ja poistaminen on mahdollista tehdä parametrisointiin perustuvalla menetelmällä. Laitemäärään on mahdollista vaikuttaa tietokantaan tehtävällä objektikuvauksilla. Tietokantaan tallennettujen JSON-dokumenttien avulla on mahdollista määritellä, kuinka paljon koneita otetaan käyttöön, ja minkälaisia ominai-

suuksia koneisiin sisällytetään. Joukko määräytyy samalla tavoin, kuin kommunikointi-instansseja kuvaava joukko (kuva 31).



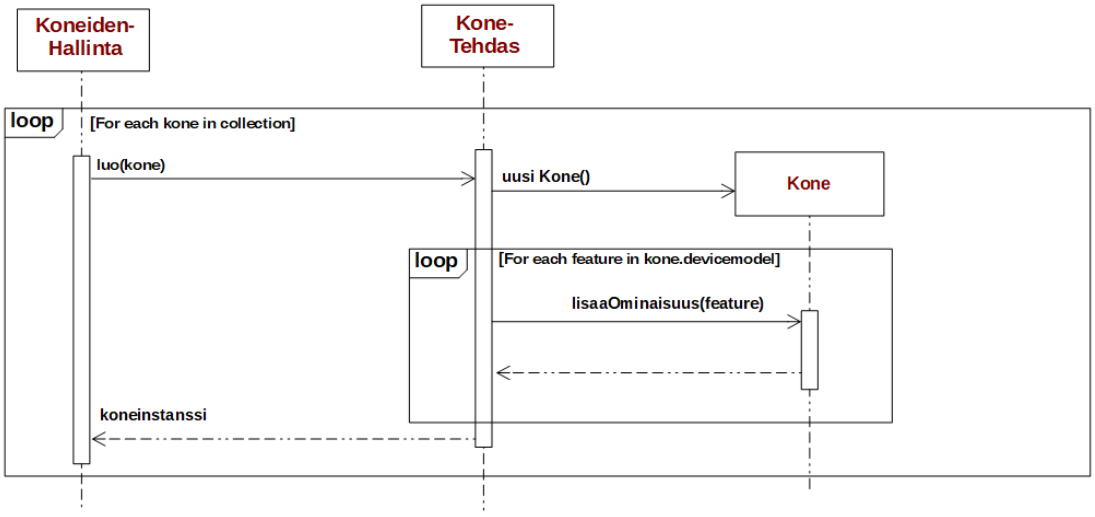
Kuva 31. *Koneet kattava joukko, jonka alta löytyvät koneita koskevat objektikuvaukset.*

Jokainen yksittäinen koneobjekti sisältää skeemaltaan samanlaisen tietorakenteen. Objektiin sisältyy joukko koneeseen sidottuja ominaisuuksia sekä koneeseen liittyvää metatietoa. Kuvassa 32 on esitetty yksityiskohtaisemmin konetta mallintava informaatiokuvaus. Kantaobjektin *Kone* alaisuudesta löytyvät kyseisen koneen id, nimi, tyyppi sekä kyseisen koneen käyttämän kommunikointi-instanssin id. Lisäksi koneobjektin alta löytyy *Devicemodel*-taulukko, jonka alaisuuteen on sisällytetty kaikki koneeseen sidotut ominaisuusobjektit. Yksittäinen ominaisuusobjekti sisältää listauksen ominaisuuteen sidonnaisista muuttujista. Muuttuja voi olla tyypiltään ristiviittausmuuttuja, eli ohjausyksikön muistialueelle viittaava (*IO Variable*) tai pelkästään ominaisuuteen sidonnainen, tietokannassa esiintyvä muuttuja (*Variable*). Ominaisuuskontekstiin sitomalla, muuttujien käyttötarkoituksella saadaan läpinäkyvyyttä, ja tarvittaessa niitä muokkaamalla koneessa esiintyviin ominaisuuksiin voidaan vaikuttaa helposti.



Kuva 32. *Koneobjektin alaisuudesta löytyvä ominaisuushierarkia*

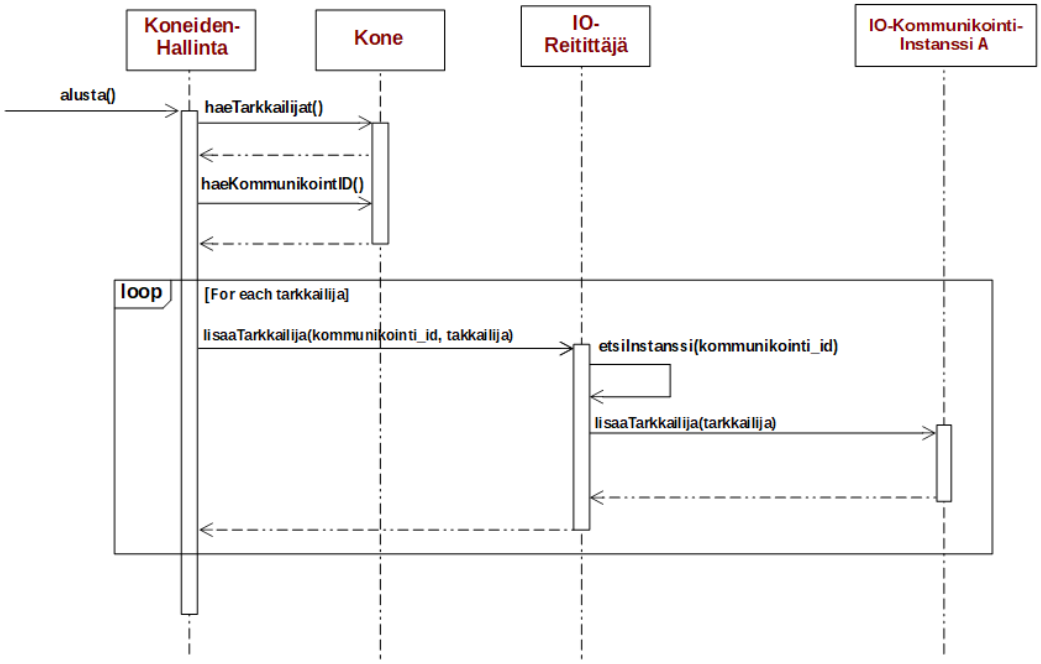
KoneidenHallinta-komponentin alaisuudessa esiintyvä *KoneTehdas* (kuva 33) rakentaa alustusvaiheessa tietokantaan määritettyjen kuvauksien pohjalta suorituksen aikana esiintyvät instanssit ominaisuuksineen.



Kuva 33. *Sekvenssidiagrammi koneinstanssien rakentamisesta ja parametrien perusteella*

Koneinstanssien rakentamisen jälkeen ominaisuuksien alta löytyvät ohjausyksikön muistialueelle viittaavat muuttujat (*IO Variable*) rekisteröityvät kuuntelemaan niille tar-

koitetun IO-Palvelun muuttujia (kuva 34). IO-palveluun viittaaminen tapahtuu ID:n perusteella. *ioservice_tagname* toimii avaimena, jonka pohjalta laiteobjektin muuttuja saadaan linkitettyä ohjausyksikön kassa kommunikoivan vastaavan komponentin muuttujalistauksen kanssa.



Kuva 34. *Tarkkailijaobjektien lisäys kommunikointi-instanssin tapahtumalähteisiin.*

Ylemmän tason komponenteilla on mahdollisuus tarkastella koneiden informaatiomallia sekä mukauttaa oma toiminnallisuutensa näiden pohjalta. Esimerkiksi aliluvussa 4.2.1 esitetty kuva 20 havainnollistaa, millä tavoin koneen ominaisuudet vaikuttavat suoraan näytelmien editointiin käytettävien toimintojen luonteeseen. Näin ollen objektikuvauksien pohjalta *NäytelmienHallinta*-komponentin ominaisuudet voitaisiin alustaa koneiden tukemien ominaisuuksia mukaillen. Samalla tavoin käyttöliittymästä on mahdollista tehdä dynaamisesti tuettuja ominaisuuksiin nähden mukautuva.

Edellisessä luvussa lokitoiminnallisuuden modularisoimiseen ehdotettuja menetelmiä olivat aspektipohjainen ohjelmointi sekä Koristelija-suunnittelumallin hyödyntäminen. Demo-sovelluksessa käytetty kieli oli Typescript, joka mahdollistaa vahvemmin tyypitetyn Javascript-koodin kirjoittamisen. Samalla se tarjoaa paremman tuen olio-pohjaiselle ohjelmoinnille. Typescript ei varsinaisessa mielessä tue aspektipohjaista sovelluskehitystä, jolloin lokitoiminnallisuuden modularisointia testattiin koristelija-suunnittelumallin avulla.

Käytännön testissä koristelijan avulla lokitoiminnallisuuspiirre käärittiin koneiden rakennusvaiheen yhteydessä kone-luokan instanssin ympärille. Konetta koskevan objektikuvauksen yhteyteen lisättiin muuttuja ”advanced_logging”, jonka avulla kone-luokan rajapinnan käytöstä lokia keräävä toiminnallisuus oli mahdollista aktivoida ehdollisesti.

4.3.2 Arkkitehtuurin modulaarisuus

Demo-sovelluksessa arkkitehtuurin modulaarisuutta edistettiin soveltamalla luokkien välisissä riippuvuussuhteissa riippuvuuksien syöttö -suunnittelumallia. Menetelmää testattiin InversifyJS kirjaston avulla, joka on Typescript-ohjelmointikielelle tarkoitettu IoC-säiliö [73]. IoC-säiliölle määritetään säännöt sille, mihin rajapintaan sidotaan mikäkin instanssi, jolloin saadaan riippumattomuus luokkien instanssien luomisesta. Kuvassa 35 havainnollistetaan, millä tavoin luokkien välinen instanssien syöttö tapahtuu. Kysymyksessä on rakentajan kautta tapahtuvasta syöttämisestä, jolloin riippuvuudet syötetään ketjumaisesti ohjelman käynnistymisen yhteydessä.

```
@injectable()
class KoneidenHallinta implements IKoneidenHallinta {

    private konetehdas: IKoneTehdas;
    private ioreitittaja: IIOReitittaja;
    private datapalvelu: IDataPalvelu;


    constructor(
        @inject(TYPES.IOreitittaja) ioreitittaja: IIOReitittaja,
        @inject(TYPES.DataPalvelu) datapalvelu: IDataPalvelu,
        @inject(TYPES.KoneTehdas) konetehdas: IKoneTehdas,

        ) {

        this.konetehdas = konetehdas;
        this.ioreitittaja = ioreitittaja;
        this.datapalvelu = datapalvelu;

    }
…
```

***Kuva 35**. Rakentajan kautta tehtävä riippuvuuksien syöttö.*

Riippuvuuksien syöttö mahdollistaa yksittäisten luokkien eristämisen ja näin ollen, esimerkiksi testien tekemisen helpommalla tavalla. Muuttamalla rajapintaan sidottavan luokan tyyppiä on mahdollista toteuttaa helposti komponentin variaatio. Esimerkiksi integraatiotestauksen osana, rajapinnan sidontaa muuttamalla olisi mahdollista vaihtaa todellisen palveluinstanssin tilalle toiminnallisuutta matkiva *Mock*-olio. Kuvassa 36 on esitetty esimerkki, miten tällaisen *Mock*-olion vaihtaminen onnistuu. Esimerkissä OPC UA-palvelukomponentin tilalle vaihdetaan IO-kommunikointia simuloiva komponentti.

```
//Alkuperäinen, todellisuudessa ohjausyksikön kanssa kommunikoiva komponentti
whisperContainer.bind<IIOService>("IIOService").to(opcuaservice).whenTargetNamed("opcuaservice");

//Tynkä, jonka avulla kommunikointia voidaan simuloida
whisperContainer.bind<IIOService>("IIOService").to(mockioservice).whenTargetNamed("opcuaservice");
```

***Kuva 36.** Rajapintojen sitominen toteutuksiin IoC-säiliökonfiguraatiossa.*

4.4 Tuoterunkoon siirtyminen

Tässä aliluvussa pohditaan vaihtoehtoja tuoterunkoarkkitehtuuriin siirtymiselle. Tuoteperheen nykyiselle sovellusrungolle on tunnistettu rajoitteita, jonka vuoksi tuoterunkoarkkitehtuuri on ehdotettu nykyisen lähestymistavan tilalle. Muutoksen tulisi kuitenkin olla taloudellisessa mielessä perusteltua, jotta siirtyminen voitaisiin nähdä realistisena. Siirtymisen suunnittelussa otetaankin tämän vuoksi tarkastelun alle mahdollinen tuoterungon investoinnin tarve.

Aiemmin aliluvussa 2.3 esitettiin mahdollisia tuoterunkoarkkitehtuuriin siirtymästrategioita. Vaihtoehtoja olivat proaktiivinen, ekstraktiivinen sekä reaktiivinen eli inkrementaalinen siirtymätapa. Proaktiivisessa menetelmässä kaikki tarvittavat resurssit luodaan heti alussa ja menetelmään sisältyy suurin taloudellinen riski. Ekstraktiivinen edellyttäisi vanhan sovelluksen hyötykäyttöä ja reaktiivisessa menetelmässä tuoterunko luotaisiin inkrementaalisesti.

Ottaen huomioon, että kysymyksessä on olemassa oleva tuoteperhe, jolle löytyy olemassa oleva sovellusrunko, voisi potentiaalisen siirtymästrategian ajatella olevan ekstraktiivinen. Nykyisessä sovellusrungossa yksittäiset ominaisuudet kuitenkin läpileikkaavat hyvin voimakkaasti toisiaan, jolloin sitä tulisi muokata vastaamaan paremmin tässä työssä tunnistettua variaatiota. Tällöin keskiöön nousisi olemassa olevan lähdekoodin refaktorointi. Refaktorointi tarkoittaa terminä sitä, että lähdekoodin ylläpidettävyyttä sekä luettavuutta parannetaan korjaamalla jo tehtyjä ratkaisuja niin, että olemassa oleva toiminnallisuus ei muutu [74, luku 1]. Refaktorointiin liittyvä merkittävin riski on se, että sitä jouduttaisiin kohdesovelluksen tapauksessa tekemään paljon, jolloin se voisi tarkoittaa koko lähdekoodin uudelleenkirjoitusta.

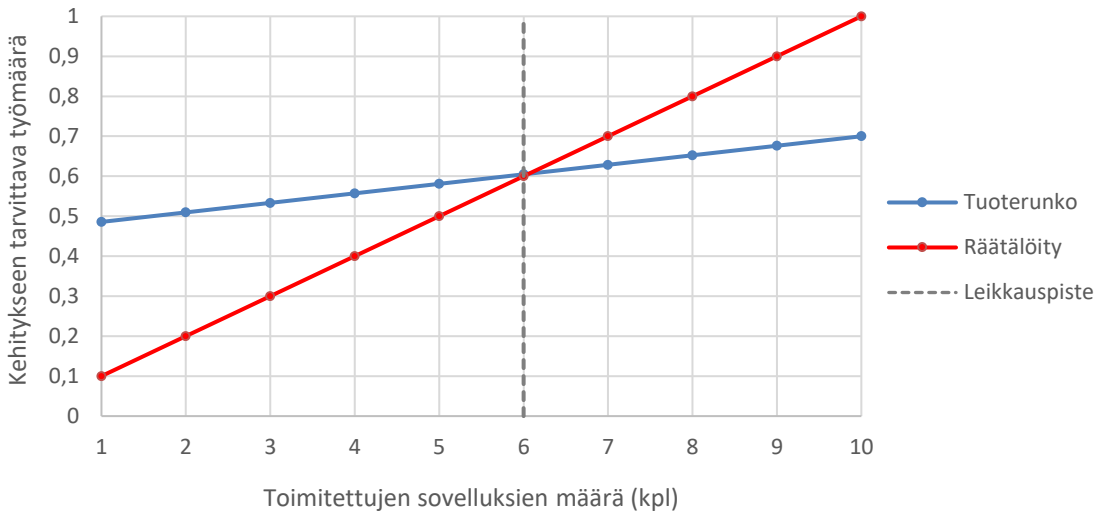
Proaktiivisessa menetelmässä kaikki resurssit luotaisiin käytännössä puhtaalta pöydältä, ennen yhdenkään tuotteen julkaisua. Kuten aikaisemmin aliluvussa 2.3 todettiin, proaktiivinen eli *Big Bang* -ratkaisu vaatii menetelmistä suurimman alkuinvestoinnin määrän. Kohdesovelluksen tapauksessa proaktiivisen menetelmän kustannuksille laskettiin aliluvussa 2.2 esitetyn SIMPLE-laskentamallin avulla alustava kustannusarvio (kaava 1). Vertailuestimaatti on määritetty jo toimitettujen, ominaisuuksiltaan homogeenisten sovelluksien edellyttämästä keskimääräisestä kehitystyömäärästä. Tuoterungolle esitetyt mahdolliset kuluestimaatit on arvioitu aikaisemmin toteutettujen teatterijärjestelmämodernisointien työmääriin pohjautuen. Arvot on normalisoitu välille [0...1] kaavaan 4 mukaisesti ja ne on koostettu taulukkoon 1.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \qquad (4)$$

Taulukko 1. Proaktiivisessa laskennassa käytetyt arvot.

Muuttuja	Normalisoitu arvo
CE_{org}	0,01
CE_{cab}	0,48
CE_{unique}	0,00
CE_{reuse}	0,02
CE_{prod}	0,10

CEorg on arvio tuoterungon vaatimasta koulutusmäärästä. *CEcab* on arvio resurssialustan rakentamiseen tarvittavasta työmäärästä, joka muodostaakin merkittävimmän kustannuksen. *CEunique* on tuotekohtaisten erikoisominaisuuksien vaatima summa, joka tässä tapauksessa jätetään tyhjäksi, sillä tuotteiden oletetaan olevan siltä osin homogeenisia. *CEreuse* on arvioitu tavoitetila yksittäisen tuotteen rakentamiselle tuoterungon pohjalta, joka tässä tapauksessa on määrältään noin viidesosa nykyisen lähestymistavan vaatimasta työmäärästä. *CEprod* esittää tämän hetkisen menetelmätavan edellyttämän kehitystyömäärän.



Kuva 37. *Estimaattien pohjalta arvioitu leikkauspiste, jonka jälkeen tuoterunko on kannattavampi verrattuna räätälöityyn menetelmään.*

Kuten kuvaajasta 37 nähdään, proaktiivinen tuoterunkoon siirtymisen edellyttämä alkupääoman tarve tarkoittaa, että tuoterungon hyödyt realisoituvat tässä tapauksessa kuuden toimitetun tuotteen jälkeen.

Soveltamalla reaktiivista siirtymämenetelmä osa tuoterunkoarkkitehtuurin resursseista tuotettaisiin inkrementaalisesti. Tuoteperheen analyysin kautta mallinnettua piirremallia tarkastelemalla, inkrementaalisuus saavutettaisiin toteuttamalla ensimmäisissä projekteissa ainoastaan pakolliseksi määritetyt ominaisuudet. Lopullisessa toteutuksessa, esi-

merkiksi koneisiin sidottujen ominaisuuksien variaatio aiheuttaa myös käyttöliittymäkerroksen komponenteissa variaatiota. Reaktiivisessa lähestymistavassa inkrementaalisuus saavutettaisiin näin ollen toteuttamalla ensimmäiseen julkaistavaan sovellukseen ainoastaan siihen tapaukseen tarvittava ominaisuuksiin sidonnainen käyttöliittymätoiminnallisuus. Lisäksi ohjausyksiköiden muutoksesta aiheutuvan kommunikointistrategian muutos voitaisiin toteuttaa inkrementaalisesti.

Ektraktiivisen sekä reaktiivisen soveltaminen yhdessä voisi myös olla toimiva ratkaisu. Näin ollen osa resursseista louhittaisiin olemassa olevista sovelluksista. Esimerkiksi vanhaa koodia voitaisiin hyväksi käyttää niin, että jo toimivaksi todettuja komponenttikirjastoja otettaisiin tarvittaessa käyttöön. Lisäksi sovelluksen käyttötapauksia mallintamalla, sekä suorituskykyä analysoimalla voitaisiin luoda resurssipohjaa sovelluksen vaatimuksista.

5. TULOSTEN ARVIOINTI JA JOHTOPÄÄTÖKSET

Työssä piirremallinnuksen kautta tunnistettiin tuoteperheen sovelluksissa esiintyvä merkittävin variaatio. Kuutta jo toimitettua ohjaussovellusta vertailemalla, ohjausjärjestelmän sekä laitteiston havaittiin aiheuttavan sovelluksissa esiintyvän merkittävimmän variaation. Analyysin pohjalta tunnistetulle variaatiolle esitettiin tämän jälkeen yksityiskohtaisia suunnitteluratkaisuja sekä variaation huomioon ottava liiketoimintalogiikkaa sekä tiedonsiirtokerrosta käsittävä korkean tason arkkitehtuuri.

Ohjausyksikön muutoksesta aiheutuvaan kommunikointityypin variaatioon ehdotettiin kommunikointimenetelmän kerroksellista abstrahointia, jossa kommunikointikomponenttien vaihtaminen olisi mahdollista toteuttaa ilman koko sovelluksen läpileikkaavaa muutostyötä. Demo-toteutuksessa käynnistyksen yhteydessä alustettavien kommunikointi-instanssien tyyppeihin ja määriin oli mahdollista vaikuttaa tietokantaan aseteltavilla parametreilla. Variaation sitominen realisoitui demo-toteutuksessa sovelluksen käynnistyksen yhteydessä. Dynaamisessa variaation sitomisessa on huomioitava, että sitominen kohdistuu oikeanlaiseen kommunikointikomponenttiin. Tällöin kantaan tallennettavan tyyppiparametrin ja tätä tukevan kommunikointikomponentin välillä tulisi esiintyä johdonmukainen yhteys. Lisäksi menetelmää tulisi kehittää siltä osin, että ainoastaan projektissa tarvittavat kommunikointikomponentit käännettäisiin mukaan osaksi lopullista sovellusta. Eli mukaan tulisi ottaa käännöstä edeltävän variaation mahdollistava työkalu.

Ohjausjärjestelmää sekä koneita koskeva variaation hallinta perustui koneita sekä laitteita kuvaaviin objektikuvauksiin, joiden ominaisuuksia muuttamalla oli mahdollista vaikuttaa koneita koskevaan variaatioon. Objektikuvauksien syntaksi noudatti JSON-skeemaa. Kuvausten pohjalta oli mahdollista vaikuttaa sovelluksen käynnistyksen yhteydessä alustettavien koneinstanssien määrään sekä muotoon. Yksittäinen konetta kuvaava objektimäärittely sisälsi joukon koneeseen sidottuja ominaisuusobjekteja, joiden sisälle oli koostettu kyseiseen ominaisuuskontekstiin liittyvät, ohjausyksikön muistialueeseen kohdistuvat muuttujaviittaukset. Lisäksi objektikuvauksiin oli mahdollista määrittää koneeseen sidonnaista metatietoa. Koneita koskevassa variaatiossa informaatiomallintamisessa syntaksi ja toteutus oli tehty työhön räätälöidysti. Vaihtoehtoisesti saman ratkaisun voisi toteuttaa esimerkiksi OPC UA:n mahdollistaman informaatiomallinnuksen keinoin [75]. Tällöin osa kuvan 26 esittämä arkkitehtuurista voitaisiin vaihtoehtoisesti korvata OPC UA:n avulla.

Sovelluksen merkittävin liiketoimintalogiikkaa rakentuu koneiden tukemien ominaisuuksien ympärille, jolloin näiden ominaisuuksien läpinäkyvyys on sovellusperheen tapauksessa tärkeää. Koska toiminnallisuuteen pystytään objektikuvaksia muuttamalla vaikuttamaan, saadaan laitteiden ominaisuuksiin sekä määriin vaikutettua tehokkaasti, kehittämällä objektikuvausten generoimiseen työkalu, yksinkertaisimmillaan vaikka taulukkolaskentaohjelma, joka mahdollistaa ominaisuuksien valinnan ja generoi tämän pohjalta JSON-skeeman. Tätä kautta koneiden muutoksesta aiheutuva tuotekohtainen varioitavuus olisi mahdollista toteuttaa tarvittaessa automaattisesti. Jatkokehitysmielessä jää myös pohtia, millä tavoin koneiden käyttöliittymään aiheuttama variaatio tulisi valjastaa informaatiomallinnuksen pohjalta. Vaihtoehtona olisi esimerkiksi suorittaa käyttöliittymää koskevan koodin generointi tuettujen ominaisuuksien pohjalta tai sitten mukauttaa käyttöliittymäobjektit aktivoitujen ominaisuuksien pohjalta dynaamisesti.

Lokitoiminnallisuuden osalta variaation toteuttamiseksi ehdotettiin aspektipohjaista sovelluskehitystä sekä ehdollisesti käyttöön otettavaa koristelija-suunnittelumallin mukaista ratkaisua. Typescript-kielellä aspekteja muistuttava toiminnallisuus olisi ollut saavutettavissa myös Typescriptin tarjoaman decorator-ominaisuuden avulla [76]. Parametrin avulla ehdollisesti käyttöönotettavaa Koristelija-suunnittelumallia käytettiin tässä tapauksessa läpileikkaantuvan ominaisuuden modularisoinnissa. Mikäli lokitoiminnallisuus haluttaisiin liittää osaksi muitakin luokkia tai komponentteja, tarvitsisi näille kaikille kuitenkin luoda omat lokitoiminnallisuus-koristelijat. Aspektipohjaisen ohjelmistokehityksen avulla läpileikkaavan ominaisuuden toteutuksesta saataisiin vielä astetta modulaarisempi, sillä lokitoiminnallisuutta suorittava aspekti tarvitsi luoda vain kerran [77].

Siirtymisstrategian suunnittelussa tuoterunkoarkkitehtuurin siirtymiselle pohdittiin erilaisia vaihtoehtoja. Ektraktiivinen menetelmä edellyttäisi nykyisen koodirungon refaktorointia. Proaktiivisessa menetelmässä sovellus luotaisiin alusta asti uudestaan ja sen alkuinvestoinnin määrä arvioitiin käyttäen hyväksi SIMPLE-laskentamallia. Alkukustannukselle laskettiin arvio, joka oli nykyiseen lähestymistapaan verrattuna noin viisinkertainen. Leikkauspisteeksi todettiin kuusi toimitettua sovellusta, jonka jälkeen tuoterungon käyttäminen olisi kustannustehokkaampaa nykyiseen lähestymistapaan verrattuna. Kirjallisuuslähteissä tyypilliseksi leikkauspisteeksi on todettu noin kolme tai neljä sovellusta [7, 11]. Soveltamalla reaktiivista, eli inkrementaalista lähestymistapaa arvioitu leikkauspiste olisi teoriassa mahdollista saavuttaa pienemmällä tuotemäärällä sekä pienemmällä riskillä. [6, 78] Inkrementaalisella tavalla leikkauspisteen jälkeen saatava hyöty olisi kuitenkin proaktiiviseen menetelmään verrattuna pienempi.

Vaihtoehtoisesti tuoterungon tuotevalikoiman laajentamista voisi miettiä kattamaan muunkinlaisia sovelluksia, kuin ainoastaan kohteena olevia näyttämömekaniikan ohjaussovelluksia. Tällöin tulisi ottaa huomioon, että erikoisominaisuuksien kasvaessa SIMPLE-laskentamallissa uniikkeja ominaisuuksia kuvaava estimaatin arvo CE_{unique} kasvaisi myös, joka vaikuttaisi leikkauspisteen sijaintiin.

Jatkossa tuoterunkoarkkitehtuurin soveltamista tulee tarkastella myös kehitystyöprosessimalli huomioon ottaen. Teatterijärjestelmä toimitetaan kokonaisuutena vesiputousmallia [79] noudattavaa prosessia mukaillen, jonka yhtenä osana toimii työn keskiössä oleva ohjaussovellus. Jatkossa kuitenkin, mikäli tuoterunkoarkkitehtuuria sovelletaan käytäntöön, tulisi kehitystyöprosessin soveltuvuutta tarkastella tuoterukoarkkitehtuurin vaatimukset huomioon ottaen. Vesiputousmallia mukaileva prosessi toimii hyvin, mikäli sovelluksen vaatimukset pysyvät muuttumattomina koko prosessin ajan [79]. Tuoterunkoarkkitehtuurissa resurssialustan voidaan kuitenkin ajatella kehittyvän yksittäisistä toimitusprojekteista irrallisena. Näin ollen resurssien kehityksessä voitaisiin tarvittaessa hyödyntää ketteriä ohjelmistokehityksen menetelmiä, kuten esimerkiksi jatkuvaa integraatiota, jolloin uusia ominaisuuksia voitaisiin kehittää pienemmällä riskillä [80].

6. YHTEENVETO

Tämän työn tavoitteena oli selvittää, millä tavoin tuoterunkoarkkitehtuuri voisi edesauttaa näyttämömekaniikan ohjaussovelluksien arkkitehtuuria sekä parantaa tätä kautta kehitystyötä. Ongelmaa lähdettiin ratkaisemaan mallintamalla kohdesovellusperheessä esiintyvä variaatio piirremallinnuksen avulla. Mallintamisen jälkeen tunnistetulle variaatiolle esitettiin suunnitteluratkaisuja, joita testattiin myös käytännössä, toteuttamalla rajatun toiminnallisuuden sisältävä demo-sovellus. Työssä pohdittiin myös mahdollista tuoterunkoarkkitehtuuriin siirtymisstrategiaa, jonka yhteydessä laskettiin arvio proaktiivisen tuoterunkosiirtymisen takaisinmaksupisteestä käyttäen hyväksi tuoterunkoja varten kehitettyä SIMPLE-laskentamallia.

Tuoterunkoarkkitehtuurin soveltuvuutta näyttämömekaniikan ohjaussovelluksen tarpeisiin tarkasteltiin tässä työssä pitkälti variaatioon sekä sovelluksen arkkitehtuurin suunnitteluun keskittyvästä näkökulmasta. Näiden kautta tuoterunkoarkkitehtuurin soveltamiselle saatiin kuitenkin alustavaa näkemystä, sillä tuoterunkoarkkitehtuurissa hyötyjä saadaan konkretisoitua siinä vaiheessa, kun rungon pohjalta luotavien tuotteiden välillä esiintyy tarvittava määrä vaihtuvia sekä samankaltaisia piirteitä. Sovellusperheen ominaisuuksia tutkimalla saatiin mallinnettua merkittävimmän variaation kuvaava piirremalli ja lisäksi näiden piirteiden huomioonottavia suunnitteluratkaisuja ehdotettiin ja testattiin käytännössä. Jatkossa esitettyjä suunnitteluratkaisuja tulee kuitenkin tarkastella myös muidenkin kuin muunneltavuuteen liittyvien laatuominaisuuksien suhteen, kuten esimeriksi suorituskyvyn sekä tietoturvallisuuden kannalta. Tämä edellyttää, että lopullinen arkkitehtuuri on kokonaisuudessaan suunniteltu, jotta tarkasteluun voidaan ottaa kaikki mahdolliset laatuominaisuudet käyttäen hyväksi arkkitehtuurin arviointiin kehitettyjä menetelmiä.

Tulosten puolesta tuoterunkoarkkitehtuuria voidaan kuitenkin pitää potentiaalisena vaihtoehtona nykyisen käytössä olevan näyttämömekaniikan ohjaussovelluksen modernisoinnissa. Teoriassa kuuden toimitetun sovelluksen jälkeen menetelmän hyödyntäminen olisi nykyiseen menetelmään verrattuna kannattavampaa. Kokonaisuudessaan tuoterunkoarkkitehtuurin käyttöönottaminen edellyttää kuitenkin monien muidenkin näkökulmien, kuin pelkästään variaation mahdollistavien teknologisten ratkaisuiden sekä siirtymisstrategian huomioon ottamista. Kysymyksessä on kuitenkin laajasta sekä pitkän kehityselinkaaren käsittävästä kokonaisuudesta, jolloin jatkossa yksityiskohtaiseen tarkasteluun tulisi ottaa myös organisaationallisiin, kehitysprosessiin, dokumentaatioon, konfiguraation hallintaan sekä testaukseen liittyviä näkökulmia.

LÄHTEET

[1] L. Bass, P. Clements, R. Kazman, Software architecture in practice, 2nd ed. Addison-Wesley, Boston, 2003, 624 p.

[2] B.P. Douglass, Real-time design patterns: robust scalable architecture for real-time systems, Addison-Wesley, Boston (MA), 2003, 528 p.

[3] Anonymous Technical Debt All things in moderation. DevIQ, verkkosivu. Saatavissa (viitattu 24.4.2018): http://deviq.com/technical-debt/.

[4] D.C. Schmidt, Why Software Reuse has Failed and How to Make It Work for You, 1999, Saatavissa (viitattu 11.4.2018): http://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html.

[5] Jean-Louis Letouzey and Declan Whelan. Introduction to the Technical Debt Concept, Agile Alliance, verkkosivu. Saatavissa (viitattu 21.10.2018): https://www.agilealliance.org/introduction-to-the-technical-debt-concept/.

[6] K. Schmid, M. Verlage, The economic impact of product line adoption and evolution, IEEE Software, Vol. 19, Iss. 4, 2002, pp. 50-57.

[7] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Springer, Berlin, Heidelberg, 2005, 467 p.

[8] L.M. Northrop, P.C. Clements, A Framework for Software Product Line Practise, Software Engineering Institute, Carnegie Mellon University, 2012, 258 p. Saatavissa: https://www.sei.cmu.edu/productlines/frame_report/index.html.

[9] S. Apel, D. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, 2013th ed. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, 315 p.

[10] P.B. Kruchten, The 4+1 View Model of architecture, IEEE Software, Vol. 12, Iss. 6, 1995, pp. 42-50.

[11] F.v.d. Linden, K. Schmid, E. Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer, New York, Berlin, 2007, 333 p.

[12] J. Bosch, Design and Use of Software Architectures: Adopting and evolving a product-line approach, Addison-Wesley, Harlow England, 2000, 354 p.

[13] M.S. Ali, M.A. Babar, K. Schmid, A Comparative Survey of Economic Models for Software Product Lines, 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, pp. 275-278.

[14] P.C. Clements, J.D. McGregor, S.G. Cohen, The Structured Intuitive Model for Product Line Economics (SIMPLE), Software Engineering Institute, Carnegie Mellon University, 2005, 67 p. Saatavissa: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=7611.

[15] J.S. Poulin, The Economics of Product Line Development, 1997, Saatavissa (viitattu 21.9.2018): http://jeffreypoulin.info/Papers/IJAST97/ijast97.html.

[16] B. Boehm, A.W. Brown, R. Madachy, Y. Yang, A software product line life cycle cost estimation model, Proceedings.2004 International Symposium on Empirical Software Engineering, 2004.ISESE '04., IEEE, Redondo Beach, CA, USA, pp. 156-164.

[17] P. Clements, Being proactive pays off, IEEE Software, Vol. 19, Iss. 4, 2002, pp. 28-31.

[18] C.W. Kruger, Easing the Transition to Software Mass Customization, Software Product-Family Engineering. PFE 2001. Lecture Notes in Computer Science, 2002, pp. 282-293.

[19] J.F. Bastos, da Mota Silveira Neto, Paulo Anselmo, P. O'Leary, E.S. de Almeida, de Lemos Meira, Silvio Romero, Software product lines adoption in small organizations, The Journal of Systems & Software, Vol. 131, 2017, pp. 112-128.

[20] K. Koskimies, T. Mikkonen, Ohjelmistoarkkitehtuurit, Talentum, Helsinki, 2005, 250 p.

[21] F. Buschmann, Pattern-oriented software architecture: Vol. 1, A system of patterns, Wiley, Chichester, 1996, 602 p.

[22] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute, Carnegie Mellon University, 1990, 147 p. Saatavissa: http://www.floppybunny.org/robin/web/virtualclassroom/chap12/s4/articles/foda_1990.pdf.

[23] H. Gomaa, Designing Software Product Lines with UML, 29th Annual IEEE/NASA Software Engineering Workshop - Tutorial Notes (SEW'05), IEEE, pp. 160-216.

[24] N. Loughran, P. Sánchez, A. Garcia, L. Fuentes, Language Support for Managing Variability in Architectural Models, in: Anonymous (ed.), Software Composition, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 36-51.

[25] M. Krisper, C. Kreiner, Describing binding time in software design patterns, Proceedings of the 21st European Conference on pattern languages of programs, ACM, pp. 1-15.

[26] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, M. Hinchey, An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry, The Journal of Systems & Software, Vol. 91, 2014, pp. 3-23.

[27] G.J. Holzmann, The power of 10: rules for developing safety-critical code, Computer, Vol. 39, Iss. 6, 2006, pp. 95-99.

[28] L. Lobato, P. O'Leary, E. de Almeida, S. de Lemos Meira, The importance of documentation, design and reuse in risk management for SPL, Proceedings of the 28th ACM International Conference on design of communication, ACM, pp. 143-150.

[29] R. Kazman, M. Klein, P. Clements, ATAM: Method for Architecture Evaluation, Defense Technical Information Center, 2000, Saatavissa: http://lore.ua.ac.be/Teaching/CapitaMaster/Assignment/ATAM-TR.pdf.

[30] R. Kazman, L. Bass, G. Abowd, M. Webb, SAAM: a method for analyzing the properties of software architectures, Proceedings of 16th International Conference on Software Engineering, pp. 81-90.

[31] U. van Heesch, V. Eloranta, P. Avgeriou, K. Koskimies, N. Harrison, Decision-Centric Architecture Reviews, IEEE Software, Vol. 31, Iss. 1, 2014, pp. 69-76.

[32] PRODUCT LINE HALL OF FAME, The Software Product Line Conference, verkkosivu. Saatavissa (viitattu 28.3.2018): http://splc.net/hall-of-fame/.

[33] pure::variants, pure-systems GmbH, verkkosivu. Saatavissa (viitattu 3.10.2018): https://www.pure-systems.com/products/pure-variants-9.html.

[34] X. Tërnava, P. Collet, On the Diversity of Capturing Variability at the Implementation Level, Proceedings of the 21st International Systems and Software Product Line Conference - Volume B, ACM, pp. 81-88.

[35] C. Gacek, M. Anastasopoules, Implementing product line variabilities, Proceedings of the 2001 symposium on software reusability, ACM, pp. 109-117.

[36] E. Gamma, Design patterns: elements of reusable object-oriented software, Addison-Wesley, Reading (MA), 1994, 395 p.

[37] Felix Bachmann, Paul C Clements, Variability in Software Product Lines, Software Engineering Institute, Carnegie Mellon University, 2005, Saatavissa: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=7675.

[38] C. Baldwin, K. Clark, Managing in an Age of Modularity, Harvard business review, Vol. 75, 1997, pp. 84-93.

[39] Separation of Concerns Take pride in your code. DevIQ, verkkosivu. Saatavissa (viitattu 4.5.2018): http://deviq.com/separation-of-concerns/.

[40] R.C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Prentice Hall, Boston, MA, 2017, 432 p.

[41] M. Yousif, Microservices, IEEE Cloud Computing, Vol. 3, Iss. 5, 2016, pp. 4-5.

[42] A. Hunt, D. Thomas, The Pragmatic Programmer: from journeyman to master, Addison-Wesley, Reading (MA), 2000, 321 p.

[43] M. Fowler, Avoiding repetition, IEEE Software, Vol. 18, Iss. 1, 2001, pp. 97-99.

[44] R.C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003, 552 p.

[45] R.C. Martin, M. Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006, 768 p.

[46] S. Schuster, S. Schulze, I. Schaefer, Structural feature interaction patterns, Proceedings of the Eighth International Workshop on variability modelling of software-intensive systems, ACM, pp. 1-8.

[47] C. Seidl, S. Schuster, I. Schaefer, Generative software product line development using variability-aware design patterns, Computer Languages, Systems and Structures, Vol. 48, 2017, pp. 89-111.

[48] Alex Culp. The Dependency Injection Design Pattern, Microsoft, verkkosivu. Saatavissa (viitattu 16.3.2018): https://msdn.microsoft.com/en-us/library/hh323705(v=vs.100).aspx.

[49] M. Immonen, SUUNNITTELUMALLIT, Kuopion yliopisto, Tietojenkäsittelytieteen ja sov. matem. laitos, 2002, 90 p. Saatavissa: http://www.cs.uef.fi/tutkimus/Teho/Suunnittelumallit.pdf.

[50] M. Fowler, Inversion of Control Containers and the Dependency Injection pattern, 2004, Saatavissa (viitattu 18.1.2018): https://martinfowler.com/articles/injection.html.

[51] M. Fowler, Module Assembly, IEEE Software, Vol. 21, Iss. 2, 2004, pp. 65-67.

[52] Spring Framework, Pivotal Software, Inc, verkkosivu. Saatavissa (viitattu 24.4.2018): https://projects.spring.io/spring-framework/.

[53] Unity Container, Microsoft, verkkosivu. Saatavissa (viitattu 24.4.2018): https://msdn.microsoft.com/en-us/library/ff647202.aspx.

[54] J. Weiskotten, Dependency Injection: Designing loosely coupled and testable objects, Dr. Dobb's Journal, Vol. 31, Iss. 5, 2006, pp. 10.

[55] Microsoft: patterns & practices. Broker, Microsoft, verkkosivu. Saatavissa (viitattu 5.4.2018): https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648096(v=pandp.10).

[56] H. Kreger, Navigating the SOA Open Standards Landscape Around Architecture, The Open Group, 2009, 28 p.

[57] M. Fowler, J. Lewis, Microservices: a definition of this new architectural term, 2014, Saatavissa (viitattu 16.4.2018): https://martinfowler.com/articles/microservices.html#footnote-etymology.

[58] W. Cunneyworth, Table Drive Design: a development strategy for minimal maintenance information systems, Data Kinetics Ltd., 1994, 96 p. Saatavissa: http://www.dkl.com/wp-content/uploads/2016/05/DataKinetics-Table-Driven-Design.pdf.

[59] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison Wesley, Redwood City, CA, USA, 2004, 736 p.

[60] M. Fowler, Introduce Parameter Object, Refactoring, 1999, Saatavissa (viitattu 24.4.2018): https://refactoring.com/catalog/introduceParameterObject.html.

[61] T.J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering, Vol. SE-2, Iss. 4, 1973, pp. 308-320.

[62] M.Ribeiro et al., Software Families, Software Products Lines, and Dataflow Analyses, in: Anonymous (ed.), Emergent Interfaces for Feature Modularization, Springer, 2014, pp. 7-21.

[63] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (ed.), ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 1997 Proceedings, Springer, Berlin, Heidelberg, 1997, pp. 220-242.

[64] A. Rashid, J. Royer, A. Rummler, Aspect-oriented, model-driven software product lines: The AMPLE way, 2011, 470 p.

[65] Postsharp, PostSharp Technologies, verkkosivu. Saatavissa (viitattu 14.6.2018): https://www.postsharp.net/.

[66] AspectJ, Eclipse Foundation, verkkosivu. Saatavissa (viitattu 13.10.2018): https://www.eclipse.org/aspectj/.

[67] Aspect Oriented Programming with Spring, Pivotal Software, Inc, verkkosivu. Saatavissa (viitattu 25.9.2018): https://docs.spring.io/spring/docs/2.0.x/reference/aop.html.

[68] Anonymous Introduction to AspectJ, Xerox Corporation, verkkosivu. Saatavissa (viitattu 31.10.2018): https://www.eclipse.org/aspectj/doc/next/progguide/starting-aspectj.html#the-dynamic-join-point-model.

[69] C. Constantinides, T. Skotiniotis, M. Stoerzer, AOP considered harmful, 2004, pp. 1-2.

[70] C. Koppen, M. Stoerzer, PCDiff: Attacking the Fragile Pointcut Problem, 2004.

[71] K. Czarnecki, S. Helsen, U. Eisenecker, Formalizing Cardinality-based Feature Models and their
Specialization, Software Process: Improvement and Practice, Vol. 10, Iss. 1, 2005, pp. 1-25.

[72] B. Sonnino, Aspect-Oriented Programming : Aspect-Oriented Programming with the RealProxy Class, MSDN Magazine, Vol. 29, Iss. 2, 2014, Saatavissa (viitattu 19.5.2018): https://msdn.microsoft.com/en-us/magazine/dn574804.aspx.

[73] Remo H. Jansen. InversifyJS, A powerful and lightweight inversion of control container
for JavaScript and Node.js apps powered by TypeScript. InversifyJS, verkkosivu. Saatavissa (viitattu 14.9.2018): http://inversify.io/.

[74] M.C. Feathers, Working Effectively with Legacy Code, Prentice Hall PTR, United States, 2005, 456 p.

[75] UA Overview: OPC UA Data Model, OPC Foundation, verkkosivu. Saatavissa (viitattu 28.10.2018):
http://wiki.opcfoundation.org/index.php?title=UA_Overview&oldid=853.

[76] Typescript decorators, Microsoft, verkkosivu. Saatavissa (viitattu 29.10.2018):
https://www.typescriptlang.org/docs/handbook/decorators.html.

[77] Matthew D. Groves. AOP vs decorator, Cross Cutting Concerns, verkkosivu. Saatavissa (viitattu 1.11.2018): https://crosscuttingconcerns.com/AOP-vs-decorator.

[78] Y. Chen, G.C. Gannod, J.S. Collofello, A software product line process simulator, Software Process: Improvement and Practice, Vol. 11, Iss. 4, 2006, pp. 385-409.

[79] S. Palmquist, M.A. Lapham, S. Miller, T.A. Chick, I. Ozkaya, Parallel Worlds: Agile and Waterfall Differences and Similarities, Software Engineering Institute, Carnegie Mellon University, 2013, 85 p. Saatavissa:
https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=62901.

[80] M. Fowler, Continuous Integration, 2006, Saatavissa (viitattu 18.11.2018):
https://martinfowler.com/articles/continuousIntegration.html.